SAECAS: Secure Authenticated Execution using CAM-based Vector Storage

Orlando Arias, *Student Member, IEEE*, Dean Sullivan, *Student Member, IEEE*, Haoqi Shan, *Student Member, IEEE*, and Yier Jin, *Senior Member, IEEE*

***Abstract*—Authenticated execution (AE) is a security mechanism which cryptographically validates an application's code as it executes, as well as verifies its control-flow. AE provides fully local guarantees which can deliver protection for control-flow, instruction flow, and software intellectual property which makes it ideal for devices with little to no connectivity. However, we find that previous AE approaches make concessions in their implementation that severely hinder their security guarantees.**

In this paper we examine the weaknesses in previous AE approaches and why these occur. We also introduce SAECAS as a mechanism to reliably perform AE in an embedded device. We formally prove the security aspects of SAECAS, demonstrating its security capabilities. Moreover, we implement SAECAS on a RISC-V core and test it on a Terasic DE2-115 FPGA board to demonstrate its capabilities, showing that a reliable system can be made with a hardware overhead of $\approx 2\times$ when including extra SoC components and no performance impact.

***Index Terms*—Authenticated execution, secure computer architectures, embedded security, cryptosystems**

I. INTRODUCTION

As more low-power easy-to-use microcontrollers are introduced to the market, so has the number of embedded devices using them increased. Although some of these devices tend to be resource constrained they can perform important tasks. For example, smart pacemakers are life saving devices that can be configured and monitored by professionals in the field of medicine, aiding people over the world. However, as these devices become prevalent, so has the number of attacks against them. In recent years, we have seen a surge of attacks against embedded devices, disrupting their application. This was the case in 2018, when researchers demonstrated the same life saving smart pacemakers can be tampered with malicious intent to deliver potentially fatal shocks to patients [1]. As a result, there has been an ongoing effort from both industry and academia to prevent and mitigate the effects of attacks in embedded devices.

Among the presented solutions, authenticated execution (AE) promises an avenue of defending against software-based attacks. Under the authenticated execution model, both the instruction stream and the order in which it is executed are checked for integrity as they enter the CPU. This provides an anti-tampering mechanism for software, changes to the program or its control-flow would be detectable by the authenticated execution policy in place. While sharing similarities to software attestation approaches [2], [3], [4], [5], [6], [7], [8], authenticated execution does not rely on a two party system for security, with all the necessary constructs for security being present in the CPU core itself.

Central to AE is that code authenticity is achieved by means of decrypting an instruction stream as it enters the CPU. However, to ensure security, instructions must be decrypted in accordance to their predecessor. Decrypting this way ensures that changes to a single instruction in the instruction stream results in subsequent instructions seen by the processor to be incorrect. This serves as the base for a control-flow integrity policy. Authenticated execution approaches often divide software into its basic blocks, encrypting them independently of each other. Execution is only allowed to occur from the start of a basic block to its proper end, at which point the next basic block in the control-flow graph is chained.

Integral to the approach is how the encryption secrets are kept in the device, and how they are used. For example, Scylla [9] employs the same encryption secret on each encrypted basic block. Because this exposes the scheme to dictionary attacks, Scylla randomizes basic blocks by adding extra instructions that do not affect the overall function of the program. Control-Flow Carrying Code [10] deals with the uniqueness problem and key storage by using Shamir's Secret Sharing to generate the decryption key for a particular basic block at the cost of partially rewriting the binary. Lastly, Sponge-Based Control-Flow Protection [11] presents a fully linear authenticated execution framework using a series of deltas that are applied to the state of the cryptographic primitive as branches are taken.

However, as we will discuss in this paper, the concessions made by previous approaches when dealing with key management and cipher state introduce unintended weaknesses in the authenticated execution policy even when the defense has full information of how the software should operate. We will demonstrate how these approaches can be bypassed by an attacker, rendering the protections of authenticated execution void. We further introduce SAECAS as a hardware module which guarantees the resiliency of authenticated execution, with no requirements to modify the software running on the device. We implement and evaluate SAECAS as part of a RISC-V core, and demonstrate its functionality and security provisions in an FPGA board.

In short, our contributions are:

Manuscript received April 18, 2020; revised June 12, 2020; accepted July 6, 2020. This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems 2020 and appears as part of the ESWEEK-TCAD special issue.

Orlando Arias, Dean Sullivan, Haoqi Shan, and Yier Jin are with the Department of Electrical and Computer Engineering, University of Florida, USA. Email: {orlandoa,deanms,haoqi.shan}@ufl.edu, yier.jin@ece.ufl.edu

Corresponding author: Yier Jin

- A proof that previous approaches are *provably unable* to fulfill the guarantees of authenticated execution, even under ideal knowledge conditions.
- A new method, SAECAS, to perform authenticated execution, which is both resilient to attack and is able to overcome the limitations of previous approaches. SAECAS does not require source code for the monitored program and operates in unmodified binaries.
- Demonstration and evaluation of SAECAS in a RISC-V core. Our implementation can be synthesized and the bitstream can be used in a Terasic DE2-115 board. Our design incurs $\approx 2\times$ hardware overhead when including all necessary SoC components, no software overhead, and is successful at providing authenticated execution.

II. AUTHENTICATED EXECUTION

A. Principles of Authenticated Execution

To fully understand the concept of *authenticated execution* we must first look at its predecessor in *Instruction Set Randomization* (ISR). ISR was initially proposed in [12] as means of counteracting code-injection attacks which were prevalent in those days. ISR attempts to counter these attacks by changing the underlying machine code representation of instructions in an ISA. This task was accomplished by placing encrypted instructions in memory, and decrypting them as they were being executed. Randomization was achieved by changing the encryption key utilized on every program load. An attacker wishing to perform a code injection attack would need to inject instructions encrypted with the proper key, otherwise the processor would execute random or invalid instructions, causing the program to crash. Eventually, ISR was abandoned as means of countering code injection attacks in favor of the memory protection attribute write-no-execute (W$\oplus$X). That is, if a portion of memory is writable, its contents can not be used as machine code.

As a response to this new memory primitive, attackers turned to existing benign code in a program. By corrupting code pointers in the program's memory, such as return addresses stored in the stack, attackers are able to change the functionality of a program by redirecting control-flow to a string of otherwise unconnected code snippets, or gadgets, present in the application. With this, the otherwise benign application is mutated into a malicious payload.

Authenticated execution extends on the ideas behind ISR to also include the means of defending against code-reuse attacks (CRAs). During execution, both the instruction stream and control-flow are checked for authenticity. However, unlike traditional ISR approaches, authenticated execution ensures the validity of the instruction stream by decrypting instructions with respect to their predecessor. This requires a stronger cipher to be employed at the time of decryption. As a side effect, this also results in improved software IP protection, as attacks against the electronic code book style of encryption employed by traditional ISR are no longer possible. Furthermore, unlike traditional control-flow integrity approaches, authenticated execution further uses the improved ISR execution model to provide control-flow integrity.

B. Previous Approaches

1) Classic Instruction Set Randomization: Instruction Set Randomization (ISR) was originally proposed in [12] by Kc et al. with the objective of countering code-injection attacks. By using a single static key which was kept secret, authors would encrypt a binary with the idea that if code were to be injected into the address space of the running program must be encrypted with the same key lest execution of the injected code become unpredictable. Encryption and decryption was performed using a simple `xor` operation between a key stored in the binary's header and the program's instruction stream.

Approaches following the initial ISR proposal concentrated in the generation, usage, and storage of the encryption key. For example, Berrantes et al. used a one time pad (OTP) to encrypt memory [13], a method which was then extended by Portokalidis et al. to add support for dynamic libraries and key management [14]. Finally, Papadogiannakis et al. presented ASIST, a hardware-based solution which considerably reduced the performance overhead of previous software based approaches [15].

We should note that these ISR approaches were geared towards countering code-injection attacks. Eventually, this class of attack was mitigated with the use of the W$\oplus$X primitive which gave way to the development of code-reuse attacks. Since classic ISR approaches provide protections that are practically equivalent to the no-execute primitive they also fail to provide the means to defend against code-reuse attacks thus losing visibility in the research community.

It should be mentioned, however, that classic ISR approaches have made a recent upswing in usage for a different reason: software intellectual property (IP) protection. This is most prevalent in devices where the confidentiality of the software running on it must be preserved. ISR here is employed as means of concealing the actual software from someone who would wish to reverse engineer it by looking at its code. For example, the NXP LPC55S69 contains a PRINCE cryptographic module which is capable of on-the-fly decryption of instructions and data from the on-chip flash [16], [17]. The PRINCE module used in these microcontrollers can encrypt and decrypt without adding any extra latency to instruction/data fetches/stores or needing to store the unencrypted data in a temporary RAM.

2) Scylla and SOFIA: Instruction set randomization (ISR) was updated to include safeguards against code-reuse attacks independently by Sullivan et al. with Scylla [9] and Clercq et al. with SOFIA [18]. Both approaches utilize a type of ISR in which instructions are decrypted relative to their predecessor, unlike classic ISR approaches in where the decryption process is more similar to an electronic code book (ECB) or one time pad (OTP). Encryption is performed at a basic block level. This new ISR property gives rise to instruction ordering in the context of their sequential ordering, giving the foundations to control-flow integrity.

Much like classic ISR approaches, an attacker must know the encryption secret in order to perform any type of code-injection. However, unlike classic ISR approaches, a basic block must be executed from its entry point, lest instructions be improperly decrypted. In both Scylla and SOFIA, if control-flow redirects execution to a place other than the entry to a basic block, the instructions are incorrectly decrypted.

Both approaches differ in how they tackle the issue of the authenticity of instructions being executed. Scylla relies on the unpredictability of invalidly decrypting instructions causing extraneous behavior in the program making it crash, whereas SOFIA adds a signature computation on basic blocks as they are executed. The former approach requires less changes to the hardware, whereas the latter requires not only a signature computation, but also storage for the valid basic block signatures to compare. Scylla forgoes any storage needs by using the same key to encrypt all basic blocks. However, because key reuse can result in weakening the encryption, Scylla permutes the locations of functions and the basic blocks within them, as well as adds dummy instructions inside basic blocks as a way to add randomization to the plain-text instruction stream. This way, dictionary and repetition attacks against plaintexts are avoided.

3) Control-Flow Carrying Code: Control-Flow Carrying Code, or C^3, was proposed by Lin et al. in [10]. C^3 extends the ideas behind Scylla and SOFIA by adding a new key management scheme to avoid key reuse and key storage. The approach uses Shamir's Secret Sharing [19] as the basis of key generation. Shamir's Secret Sharing is based on the fact that a set of n unique points can be fit by exactly one $n-1$ degree polynomial. The scheme is used to divide a secret S into k different parts in a way that knowing only $n < k$ parts the full secret can be reconstructed. We will now cover the basics of the scheme.

Let $n, k \in GF(p)$, where p is a prime number, and $0 < n < k$. We wish to divide the secret $S \in GF(p)$ into k parts so that only n parts of the secret are needed to reconstruct it. We build a polynomial $f(x) = \sum_{i=0}^{n-1} a_i x^i$ with $a_0 = S$ and $a_i \in GF(p), a_{n-1} \neq 0$. We then compute k non-zero input points in the polynomial $(m, f(m))$ building the set $M = \{(m_j, f(m_j) | 1 \leq j \leq k\}$. Any subset of points $M_s \subset M, ||M_s|| = n$ is sufficient to obtain the original polynomial, and therefore the secret $S = a_0$. Reconstruction of the secret is done by computing $a_0 = \sum_{i=1}^{n} f(x_i) \prod_{j=1, j\neq i}^{n} -x_j(x_i - x_j)^{-1}$.

C^3 uses an $n = 3$ of k parts to compute the secret, where the parts are the source address, the destination address, and a *master key*. The latter is used to avoid a disclosure of a code pointer in the source-destination pair from revealing the secret a_0 used to decrypt the target basic block. C^3 uses these three elements to build the $y = a_0 + a_1 x + a_2 x^2 \pmod{p}$ which constructs the secret a_0. The secret is used to encrypt a particular basic block. Points (x, y) in the polynomial are obtained by decomposing addresses. On a control-flow transfer, the source and destination address are used to build two points, and thus obtain the secret a_0 which encrypts the basic block. The C^3 scheme requires multiple basic blocks that are the target of the same control-flow instruction to be aligned in such way that their addresses can become points in the same polynomial. For this purpose, the binary is rewritten and basic blocks are reallocated during the encryption pass.

4) Sponge-Based Control-Flow Protection for IoT Devices: Sponge-Based Control-Flow Protection (SCFP) was proposed by Werner et al. in [11]. Much like C^3, SCFP concentrates on key generation and management. However, unlike previous approaches, SCFP aims to keep a single running state on the cipher decrypting the program through its execution. The state is updated through *deltas* or *patches* which are added to the program around control-flow instructions. The initial state is generated from a secret key which is unknown to the attacker. Even if the attacker knows the deltas used to update the state of the cipher, without knowledge of the initial state the keys used to decrypt instructions remain unknown.

SCFP makes use of the PRINCE cryptosystem, as an unrolled implementation has low latency. Internally, PRINCE uses a sponge-based construct hence the name of the approach. Binary rewriting is required to insert the necessary patches.

C. Observation on Previous Approaches

As we will examine in detail in Section VII, previous implementations of authenticated execution are *provably* unable to enforce the security guarantees of this type of defense, even under the condition that a full control-flow graph for the program being protected is known. Traditional Instruction Set Randomization approaches are unable to enforce any kind of control-flow integrity, much like the newly introduced NXP LPC55S69 PRINCE module. Scylla and SOFIA both are only capable of enforcing a very relaxed control-flow integrity policy. Control-Flow Carrying Code allows an attacker to compute valid keys and predict the outcome of decryption. Lastly, Sponge-Based Control-Flow Protection is unable to enforce a full control-flow graph due to the way the cipher states are synchronized.

For this reason, we propose SAECAS as a secure implementation of authenticated execution under the conditions where previous approaches fail. SAECAS is unique in that it decouples control-flow checks from the instruction decryption process while still preserving a relation between the two.

III. Threat Model and Assumptions

We assume an attacker which may have access to the device. The attacker wishes to alter the device's functionality by tampering with the software running on it through either direct changes to the code, or code-reuse attacks. For example, an attacker may use memory vulnerabilities to inject program code, leak code from the running software, or attempt to compute the cryptographic secrets used to encrypt the software.

We do not require full verification of the software running on the device when the device boots, however, we do require proper verification of the stored software's metadata. For this purpose, we use an immutable root of trust, however this portion of the implementation is not strictly unique or required. Our goal with the defense is to thwart attempts to change the software on the device dynamically at runtime.

We aim to protect statically linked applications, where library routines become part of the application binary, with no underlying operating system or dynamic linker. This is the most common type of application present in embedded devices. We do not protect against inherently malicious software. Much like cryptographic algorithms, we require secure storage of the key used to decrypt the software. Furthermore, our implementation does not protect against side-channel or invasive attacks to extract cryptographic secrets. Methods to deal with this type of attack have been proposed before [20], [21], [22] and are an orthogonal avenue of research.

IV. SaeCAS Design

As previously discussed, authenticated execution provides the means to verify both control-flow and instruction-flow of a program as it is executed. This is achieved in part by encrypting instructions in the software in a way so that decryption of an instruction depends on its predecessor. However, as with other cryptosystems, key storage and usage becomes a challenge. This is exacerbated by a powerful attacker who has access to the device and can dump the non-secure areas of the firmware for analysis. Consequently, when designing the system we need to be aware of the following issues:

- CFI must be stateful: function returns must return to their caller.
- CFI policy must enforce CFG: CFI mechanism must not introduce extraneous edge to the control-flow graph.
- Keys must remain secure: encryption keys must not be in the non-secure (encrypted) instruction stream where an attacker can view them.
- Keys and decryption must be unpredictable: an attacker with partial knowledge of the system (e.g. location of some basic blocks, code pointers) must not be able to infer further information from the system.

In particular, we note that with previous schemes attempting to dynamically generate keys as a function of source-destination addresses will result in predictable decryption of instructions and a relaxed CFI policy. We also note that stateful encryption of the instruction stream on its own will not be able to provide a strong CFI policy. As such, we conclude that *encryption keys or any information used to generate encryption keys* must be kept hidden from the attacker, and that control-flow information must be kept *decoupled* from the instruction stream.

A. Design Overview

The aforementioned insights lead us to our main design considerations. We must keep control-flow information, as well as data which is used to generate the encryption keys employed to decrypt the instruction stream. We wish to perform the decryption process in a way that minimizes any latency in the CPU's pipeline, as to avoid affecting the CPU's throughput. In this same vein, updates to the cipher must be done in a single clock cycle. Furthermore, our changes to the CPU must not interfere with microarchitectural design choices used to decrease CPI while still providing all guarantees of authenticated execution.

We address the issue of storage of both control-flow information as well as seeds for encryption of basic blocks with a content-addressable memory (CAM). A CAM is a type of memory that, unlike conventional random-access memory (RAM) from which information accessed using a linear index, uses a data word as a search item returning any associated contents if the data word is found. When detecting a control-flow transfer, we send the CAM the source-destination address pair. The CAM searches for an entry which contains this source-destination pair and if found returns the secret used for the purpose of decryption. If the control-flow address pair is not found in the CAM, the system raises an exception signaling a control-flow violation. The purpose of the CAM is then twofold: it stores the metadata necessary to start the decryption of a basic block, and it decouples control-flow information from the decrypted instruction stream serving as its own CFG metadata repository.

The state of a cryptographically secure pseudorandom number generator (CSPRNG) is updated in parallel to instruction fetches. This allows for decreasing the effects on timing in the CPU's datapath. On control-flow instructions, the next state of the CSPRNG is updated with information obtained from the CAM. Data stored in the CAM is kept from being accessed by other parts of the SoC, regardless of any security state. Verification of control-flow information occurs at a later stage of the pipeline (see Section IV-D). Decrypted instructions are fed back into the cipher to update state information.

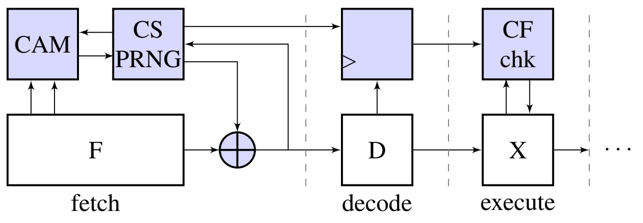


Fig. 1: SaeCAS design overview. Our additions to the CPU pipeline are highlighted. We extend the instruction fetch stage of the CPU to decrypt fetched instructions so that the current instruction is decrypted with respect to its predecessor. Initialization vectors are obtained from a content addressable memory (CAM) which is indexed using source-destination address pairs from the branch predictor. These serve as the starting point for the decryption of a basic block. If an address pair is found in the CAM, it is forwarded to the execution stage, where control-flow is verified after the branch predictor's operation is checked.

We show the overall design of our system in Figure 1 with additions to the CPU being highlighted. Our decryption stage works mostly in parallel with the instruction fetch. We only place a small amount of combinational logic between the fetch and decode stages of the CPU as to diminish the effects of decryption in the pipeline's throughput. The ensuing sections explain the portions of our scheme.

B. Instruction Key Generation and Decryption

To generate keys that decrypt the instruction stream we use a cryptographically secure pseudorandom number generator (CSPRNG) based around a block cipher. The CSPRNG yields a number which is used to decrypt a single instruction. The output of the CSPRNG and the decrypted instruction are used as feedback for the CSPRNG algorithm. We show the structure of the decryption engine in Figure 2.

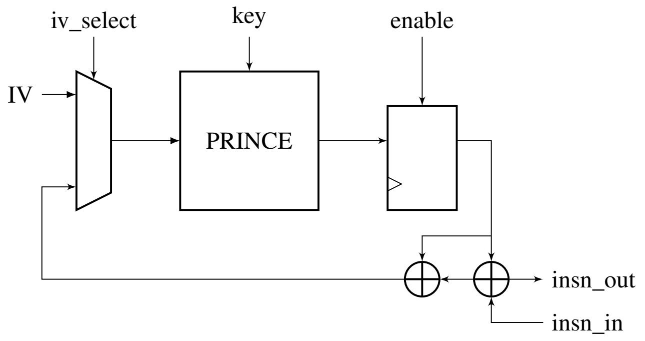


Fig. 2: Decryption key generation and process. The decrypted instruction becomes part of the next state used by the PRINCE engine. Our PRINCE implementation is unrolled, allowing for single cycle operation.

For the purposes of explaining the datapath in the CSPRNG, we will use P_k as the encryption function. To generate a decryption key D_k we utilize the lower $31\,\mathrm{bit}$ of the output of the CSPRNG. That is, $D_k = P_k(S_n) \mod 2^{32}$, then the decrypted instruction becomes $\text{insn}_{out} = \text{insn}_{in} \oplus D_k$. To compute the feedback path into the CSPRNG, S_{n_1}, we combine the current upper $32\,\mathrm{bit}$ of the CSPRNG output with the decrypted instruction. That is, $S_{n+1} = P_k(S_n \oplus (\text{insn}_d \times 2^{32}))$. We reseed the CSPRNG with a known IV whenever we start executing a basic block, that is $S_0 = P_k(IV)$.

C. Control-Flow and IV Storage

The initialization vectors (IVs) for generating decryption keys is stored using a content-addressable memory (CAM). The CAM is indexed using control-flow source-destination address pairs returning the IV as well as whether the returned data is valid or not. We show the structure of a CAM line in Figure 3.

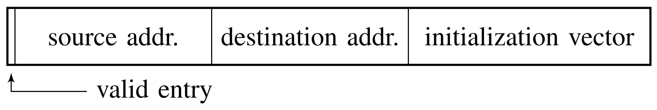


Fig. 3: SAECAS CAM line structure. Upon control-flow transfers, the CAM is indexed using source-destination address pairs to extract the seeds to the encryption secret used for the purpose of decryption. If the source-destination address pairs are not found in the CAM, the information is propagated to a later stage of the pipeline to possibly raise an exception.

The contents of the CAM are initialized using software in a boot ROM (see Section IV-E) through a set of memory mapped registers. These registers provide a write-only interface to the CAM. Boot code then locks the CAM from further writes before yielding control to the application software. This prevents memory errors in the application from being used by an attacker to inject extra data into the CAM.

On a control-flow instruction, the data obtained from the CAM is used to decrypt the next incoming basic block. The result of the valid entry is also forwarded through the pipeline to the execution stage where control-flow is verified.

D. Control-Flow Checking

The valid bit obtained from reading the CAM during a control-flow is forwarded through the CPU's pipeline until it reaches the stage where control-flow instructions are executed. The reason for this design choice is the way CPUs treat branch instructions. A shadow stack is used to preserve control-flow state on function calls and returns. The shadow stack is not memory mapped, making it inaccessible to software running on the CPU. We will cover control-flow transfers in detail in Section V-D.

E. System Startup

A non-modifiable boot ROM is used to initialize the system. The ROM contains initialization code which cryptographically verifies control-flow information, decrypts it, and stores it in the CAM. The boot ROM code locks the CAM from any future writes, as to avoid software from modifying its contents. Then, it enables the decryption subsystem and jumps into the entry point of the encrypted application.

ROM boot code is small and can be formally verified if needed. Care must be taken so that the boot code performs safe cleanup of used memory areas and CPU registers as to avoid leakage of information from the boot process. Although these details improve security of the system, the specifics of the implementation and verification of the boot code are outside the scope of this paper.

V. SAECAS IMPLEMENTATION

As part of SAECAS we extended the RISC-V ORCA core [23] to provide our authenticated execution framework. We also created helper tools to aid the collection of control-flow metadata and automate the encryption of instructions.

A. The VectorBlox ORCA Core

The VectorBlox ORCA Core is a $32\,\mathrm{bit}$ RISC-V core optimized for FPGA implementations [23]. The core is written in VHDL and is distributed under the BSD license. In its stock configuration, ORCA implements a 5 stage pipeline, with a branch predictor in the instruction fetch.

The branch predictor by default is implemented as a 16 entry branch target buffer (BTB). The BTB behaves like a direct mapped cache, indexed and tagged by the source address. If the core is configured to forgo the use of the BTB, the branch predictor behaves as a simple $\mathrm{pc} + 4$. That is, without a BTB the instruction fetch stage will always read the next instruction and forward it to the decode stage. Branches in ORCA are resolved in the execution stage. If a branch is taken and it was mispredicted, the execute stage sends the corrected program counter as well as the source address of the branch instruction to the instruction fetch stage. The contents of the fetch and decode stages are invalidated, and instruction fetches resume from the new address. If the BTB is present, the source/destination pair is stored in the BTB.

The ORCA core's reset vector is located at address `0x00000000`, however this can be configured to a different location at synthesis time. When the core starts, code begins executing from this address. The ORCA core has an option to enable vectored interrupts, although in the default configuration this is disabled.

ORCA implements three different bus masters: Avalon [24], AMBA AXI [25], and Wishbone [26] which allows for ease of integration of the core into any SoC. Our implementation uses the Avalon topology as the SoC bus, but this is not a strict requirement. We only perform bus accesses from the boot ROM to configure the authenticated execution subsystem.

B. Initialization and Boot ROM

When the core initializes, the metadata in the content addressable memory (CAM) is undefined. Bus matrix reset signal is propagated into the CAM, invalidating all its entries. This signal also unlocks write accesses to the CAM so that it can be initialized by the Boot ROM. The CAM is exposed to the SoC using a write only Avalon bus interface. That is, with the exception of the CAM status register, CAM control registers can only be written to but not read. This ensures that software can not leak any control-flow metadata.

The memory map of the ORCA core was further divided, reserving the lower $4\,\mathrm{kB}$ of memory for the boot ROM code. Since at this point in execution the decryption subsystem has not been populated, this code resides in plain text. The boot ROM can then verify encrypted control-flow and initialization vector (IV) metadata, which can be accomplished using standard digital signatures. Control-flow and IV metadata can then be decrypted and loaded into the CAM. Boot ROM code then locks CAM writes, preventing changes to its contents from memory errors in application software. The boot ROM code is trusted. The master key for the cipher, as well as the certificates are stored elsewhere in the SoC in a region only available to the boot ROM code, and are used only once to initialize the contents of the CAM and the cipher. Protection of this area is achievable by a simple check of the program counter on a memory access to this region. If the program counter lies within the boot ROM, then access is allowed and the key and certificates can be read, otherwise, zeros are returned.

C. Integrating Authenticated Execution into ORCA

Our decryption unit serves as an extension to the fetch stage of the ORCA core. No extra clock cycle is required for data propagation. An unrolled PRINCE implementation allows us to generate cryptographically secure random numbers (CSRNG) in a single clock cycle. A small instruction decoder identifies control-flow instructions. If a control-flow instruction is detected, the current program counter and the predicted program counter signals that are propagated to the rest of the pipeline are used to index into the CAM to obtain IV and valid control-flow transfer information. The IV is used to reseed the CSRNG, and the valid bit from the CAM is propagated into the pipeline.

The execute stage is modified to utilize the propagated control-flow valid information from the modified fetch stage to determine whether a control-flow transfer is valid or not. Moreover, a shadow stack is also added to account for state changes in the control-flow graph as described in [27]. Whenever a call instruction which targets a valid callee is executed, the return address is pushed into the shadow stack. Whenever a return instruction targeting a valid callee return site is executed, the last stored value in the shadow stack is popped and compared to the target program counter. If a mismatch is detected, a control-flow violation is reported.

D. Control-Flow Checking in ORCA

During execution, the instruction fetch stage of the ORCA core is constantly generating a predicted program counter which is used to fetch the next instruction. By default, the predicted program counter is the current fetch address plus one instruction word, that is $\mathtt{pc} + 4$. However, if the BTB is present, and there is an entry in the BTB that matches the current fetch address, then the BTB is used to generate the predicted program counter and the next instruction is fetched from the address obtained from the BTB. Ultimately, control-flow instructions are handled in the execute stage. This is far too late to perform decryption, as the instruction must be decoded and its operands must be obtained before it reaches the execute stage in the pipeline. If the predicted program counter generated by the instruction fetch stage was incorrect, the execute stage sends a corrected program counter to the instruction fetch stage, coupled with the source program counter. Fetched instructions due to the misprediction are invalidated in the pipeline.

For the instruction decode stage to obtain the proper operands, the decryption engine must exist between the fetch and decode stages. However, during that stage of the pipeline the computed target for a branch instruction may not be properly resolved. As such, when encountering a branch instruction our implementation relies on the predicted program counter sent by the fetch stage. The decryption logic detects whether the decrypted instruction is a control-flow instruction. If so, it performs a lookup in the CAM for the next IV using the current program counter and the *predicted program counter*. If the control-flow pair is not found, the CAM returns an IV of 0, and an invalid entry value. Otherwise, the CAM returns the corresponding IV, and reports a valid transfer which is propagated to the execution stage. The execution stage uses this information in conjunction with the operation of the control-flow instruction to decide if a control-flow violation has taken place using the following rules in addition to checks against the shadow stack:

1) If the transfer detected in the CAM is invalid, and the execute stage detected a branch misprediction, then no exception is raised. Furthermore, a corrected program counter is sent to the instruction fetch stage coupled with the source address. The decryption subsystem uses this information perform a new lookup in the CAM starting the process anew.
2) If the transfer detected in the CAM is valid, and the execute stage detected a branch misprediction, then no exception is raised. Handling of the branch correction follows the same process as above.

3) If the transfer detected in the CAM is invalid, and the execute stage detected *no* branch misprediction, then *we do not commit the instruction and an exception is raised*. This is because the software has taken a branch to a location which was not intended as no record of such control-flow is present in the CAM.
4) If the transfer detected in the CAM is valid, and the execute stage detected a proper branch prediction, no exception is raised. Moreover, instructions that entered the pipeline as a result of the proper branch prediction have been properly decrypted since the proper IV was loaded into the CSRNG and no extra action is taken.

These rules allow for authenticated execution to take place with the same pipeline throughput as an unmodified pipeline. There is no need to stall the pipeline waiting for a control-flow instruction to resolve in the execution stage. The decryption process to continue until control-flow can be verified at the time of instruction execution, which allows software to run without loss of performance.

E. Metadata Generation and Encryption

We added a back-end pass to the LLVM compiler infrastructure [28] to streamline the collection of control-flow metadata. The back-end pass records control-flow instructions and their targets when possible and instruments the source and destination pairs using symbolic labels. Our pass also stores these symbolic labels in a new non-loadable sections of the binary. At link time, the labels in the section are populated with the addresses of the labels in program code.

We encrypt our binary with a Python script which we run after linking has completed. Basic block information is extracted from the newly created section and generating the contents of the CAM at this point. The Python script creates a new binary which contains the CAM data. This area is used by the boot ROM's code to initialize the CAM before control is passed to the encrypted binary.

VI. Experimental Results

We tested SaeCAS on the Terasic DE2-115 board, which uses an Intel Cyclone IV EP4CE115 FPGA. We now discuss hardware and software overhead of our implementation, as well as power considerations.

A. Hardware Evaluation

Our hardware implementation is written in 1319 lines of VHDL, not including any SoC glue logic. Table I show details on hardware overhead when synthesized targeting the Cyclone IV EP4CE115. These values include an additional SoC infrastructure needed to support our mechanism, including bus logic and additional address decoders.

Most of the overhead is due to the CAM and the unrolled PRINCE implementation. The ORCA core in the DE2-115 board uses a 100 MHz clock. Our additions to the ORCA core and associated SoC do not affect the timing requirements.

TABLE I: Resource usage of SaeCAS in a Cyclone IV EPC4CE115 FPGA. Our implementation has total hardware overhead of $\approx 2\times$ in our test platform.

	ORCA SoC	ORCA SoC+SaeCAS
Flip-Flops	8839	18550
Logic Elements	14799	32442
Block RAM Bits	146960	147760

B. Hardware Overhead of Different Subsystems

We show a breakdown of the overhead of different components of our design in Table II. As expected, most of the datapath overhead lies in the implementation of the PRINCE cipher, whereas the CAM circuitry contributes the most to the area overhead. Results were obtained using Intel Quartus Prime 19.1.0 Build 670 targeting a Cyclone IV EP4CE115F29C7 FPGA at the most aggressive levels of optimization.

TABLE II: Per-subsystem overhead for our implementation in terms of flip-flops (FF), logic elements (LE), block RAM bits (BRAM), and maximum frequency (F_{max}). The highest contributors to area overhead and datapath delays are the CAM and the unrolled PRINCE implementation, respectively.

Component	**FF**	**LE**	**BRAM**	$\mathbf{F_{max}}$
PRINCE CSRNG	64	1852	0	100 MHz
CAM	9307	15630	0	> 100 MHz
Shadow Stack	52	69	1400	>100 MHz

We tested our unrolled PRINCE implementation targeting a 100 MHz clock with a fixed key. Faster clock speeds result in the design suffering from negative slack. We configured the CAM to utilize 14 bit for source and destination addresses, a 64 bit IV field, and 100 lines. This allows us to address 64 kB of code, and to store 100 unique control-flow address pairs. Our shadow stack implementation contains a total of 100 14 bit entries. The chosen values are sufficient to hold all control-flow data for BEEBS [29].

We should stress that the reported numbers target an FPGA implementation of the system. There are further optimizations that can be made to our design when targeting an ASIC platform. For example, the `xor` gates in the PRINCE cryptosystem used alongside the round constants can be replaced with inverters, which further reduces propagation delays in the cipher's datapath. In an FPGA-based platform, the synthesis of `xor` gates have the same delay cost as the synthesis of inverters, as logic functions are implemented using SRAM-based lookup tables inside logic blocks. We defer testing on an ASIC-based platform as future work.

C. Software Overhead

We utilized our tools to compile BEEBS [29], a series of embedded benchmarks, and the RISC-V testsuite and run them on the modified ORCA core. Unsurprisingly, we do not exhibit any overhead issues when running software. Our binaries are not instrumented with extra instructions or data which is used by software in runtime, and our hardware implementation does not interfere with the behavior of the different stages of the pipeline. Moreover, we do not alter the placement of instructions or data sections in memory. As such, we do not encounter any software overhead.

D. Overhead of Other Approaches

We compare SAECAS to previous authenticated execution implementations in Table III in terms of hardware and software overhead, and effects on the datapath.

TABLE III: Overhead comparison between SAECAS and previous authenticated execution approaches. We show comparable or better overhead in terms of hardware (HW), software (SW), and clock frequency reduction due to introduced datapath delays (CLK). Hardware overhead is with respect to the baseline platform.

Approach	Overhead		
	HW	SW	CLK
Scylla [9]	-†	20%	-†
SOFIA [18]	1.12×	149%	−23.2%
Control-Flow Carrying Code [10]	-†	70%	-†
S.B. Control-Flow Protection [11]	1.32×	9.1%	100 MHz‡
SAECAS (this work)	≈ 2×	0%	100 MHz

† No hardware implementation available
‡ Reported as target frequency

The authors of SOFIA used the RECTANGLE-80 block cipher [30] unrolled in the critical path of a LEON3 SPARCv8 processor [31]. An implementation using PRINCE is also presented. Most of the reduction in clock speed stems from using the cipher this way [18]. This is much unlike our design, where the bulk of the cryptosystem operates in parallel to the fetch stage of the CPU. The authors report a hardware overhead of $\approx 1.23\times$, but it is unclear if this is relative to the LEON3 core or a baseline SoC, if extra storage considerations were accounted for in this overhead, and whether the RECTANGLE-80 or PRINCE version were evaluated.

Neither Scylla [9] or Control-Flow Carrying Code [10] provide a hardware implementation. The former opts to use its own execution framework and the latter uses the Intel PIN dynamic instrumentation tool [32] to provide emulation platforms for their respective execution environments. The reported software overhead for both approaches are relative to executing binaries in their respective execution wrappers.

Lastly, SCFP requires binary rewriting in terms of adding the necessary deltas to the application code. This leads to a reported software performance overhead of 9.1%. Much like our design, the cipher operates in parallel to the processor's datapath, which introduced no changes in the operating speed of the design. As such, the authors report that the target frequency of their core remained unchanged at 100 MHz [11], with no further data given. Unfortunately, the authors do not make it clear whether the presented hardware overhead of $1.32\times$ is with respect to the core they used as part of their implementation, or an SoC.

In comparison, our design shows similar albeit higher hardware overhead while providing stronger security guarantees. Moreover, we incur no software overhead, as we are capable of running *unmodified* binaries in an SoC whose datapath remains virtually unchanged. Cipher operations in our design occur in parallel to the fetch stage of the processor, and other introductions to the processor's datapath barely have any effects on delays.

E. Usage of Different Cryptosystems

The cryptosystem used as part of the decryption scheme will have an overall effect in the throughput of the system. For our purposes, we utilized the PRINCE cipher, which was created specifically for unrolled applications in embedded devices [33]. The cipher also commercially used by NXP Semiconductor in their LPC55S69 series of microcontrollers as a way to provide low latency runtime decryption of a program stored in the internal flash memory [16], [17]. Our usage of PRINCE stems from its creation goals, and that to the best of our knowledge only attacks on reduced round versions of the cipher have been published [34], [35].

An implementation of SOFIA [18] used the RECTANGLE-80 cipher [30] to detrimental effects in the CPU's datapath. However, the unrolled cipher was used as part of the datapath itself. The authors show that using PRINCE in this particular way still affects the overall clock speed of the resulting processor, but to a much lesser extent.

We believe that the use of other ciphers with multiple rounds, such as AES-128, in an unrolled fashion would reduce the frequency at which the overall system operates to avoid stalling the CPU's pipeline. Alternatively, a sequential version of the cipher could be used, but at a higher clock speed.

F. Discussion of Power Considerations

Since we target IoT devices, it is only fair we include a discussion on power considerations. IoT devices by their nature are constrained, and some have low power consumption requirements. CAMs by their nature tend to be relatively power hungry. However, there are methods to reduce power consumption in CAM circuitry [36], [37], [38].

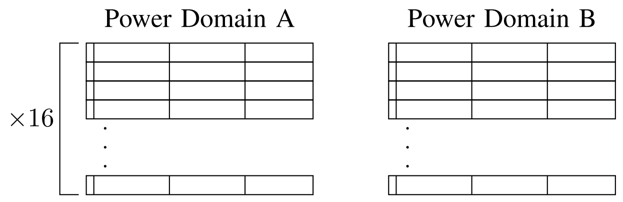


Fig. 4: Dividing the CAM into power domain groups allows for the shutdown of unused CAM regions lowering power demands on smaller programs.

Analysis of BEEBS gives an almost linear relation between program size and control-flow pairs, suggesting that smaller applications would not make use of much of the CAM's capacity. Even with the application of the aforementioned methods, we are left in situations where power is wasted in unused CAM resources. To optimize CAM and power usage we propose dividing the CAM into different software-controlled power domains.

In Figure 4 we show an example of the CAM divided into discrete domains, each containing 16 lines. During initialization, the boot ROM can enable the necessary CAM regions to load all control-flow and IV metadata. Unused areas would remain off for the duration of the device's operation. Since implementing and testing such mechanism can only happen in ASIC design, we leave this as future work.

VII. Security Considerations

A. Security Analysis of SaeCAS

We tested the security of our mechanism by inserting a vulnerability in an otherwise benign binary that allowed us to alter control-flow and modify the instruction stream. This is in line with the attacker model described in Section III. We ran our binary in the DE2-115 board using our modified ORCA core and attempted to deploy a code-reuse attack, a code injection attack, and leakage of code.

As expected, our system was able to detect deviations from the control-flow graph. Since the boot ROM locks the CAM before jumping into encrypted code, we were unable to use the memory vulnerability to add new control-flow and IV metadata. We were also unsuccessful in deploying a code injection attack. This is because the code needs to be encrypted with the proper IV and key. Following the attacker model, the key is not available to us, and with the CAM being write-only we are unable to recover the necessary IVs for encryption. We were able to use the memory vulnerability to recover encrypted code, but obtaining the plain-text was impossible given our lack of knowledge from the encryption key and IVs used.

B. Proof of Security

We assume a full control-flow graph for the program running on a device which implements SaeCAS. We let $\mathbb{A}$ be the set of addresses in the program's instruction space. We let $\mathbb{A}'$ be the source of genuine instructions in the program. We also let $\mathbb{S} \subseteq \mathbb{A}'$ be the set of valid source addresses, and $\mathbb{D} \subseteq \mathbb{A}'$ be the set of valid destination addresses.

We let $C : \mathbb{S} \times \mathbb{D} \rightarrow \{0, 1\}$ be a function that determines whether a control-flow address pair is invalid (0) or valid (1). We let $\mathbb{E}$ be the set of entries in the CAM. An entry is valid if its valid bit is set. We define the function $V : \mathbb{E} \rightarrow 0, 1$ representing valid entries in the CAM. However, the CAM in SaeCAS contains all valid source-destination pairs of all control-flow transfers. That is, for entry i, $V(i) = 1 \leftrightarrow C(S_i, D_i) = 1$. Then, $\forall S \in \mathbb{S}, \forall D \in \mathbb{D}, C(S, D) = 1 \rightarrow V_{S,D} = 1$, meaning that only valid source-destination pairs are allowed in control-flow transfers. Conversely, $\forall S \in \mathbb{S}, \forall D \in \mathbb{D}, V_{S,D} = 0 \rightarrow C(S, D) = 0$. That is, all invalid control-flow transactions are implicitly stored in the CAM. Moreover, as proven by Jin et al. in [27], it is insufficient to track valid edges in a control-flow graph for full control-flow protection, introducing the concept of state in the control-flow graph and demonstrate that a policy must enforce software execution state. SaeCAS achieves this by implementing a shadow stack of return addresses.

Proof for proper decryption follows the state required of the cipher. A basic block b is decrypted using key k and initialization vector IV. The first required state for decryption on a basic block becomes $S_{b,0} = P_k(IV)$, and subsequent states are computed by performing the operation $S_{b,n+1} = P_k(S_n \oplus (\text{insn}_{b,n,d} \times 2^{32}))$ where P_k is the PRINCE function with key k. For decrypting the i^{th} instruction of the basic block we utilize $D_k = S_k \mod 2^{32}$. Upon a valid control-flow transfer, the CAM returns IV for the target basic block. An invalid control-flow transfer returns no valid data, as previously proven. Hence, $S_{b',n'} = P_k(S_l \oplus (\text{insn}_{b-1,l,d} \times 2^{32}))$, where $\text{insn}_{b-1,l,d}$ is the last decrypted instruction of the parent basic block, and $S_{b',n'}$ is the state which will be used to decrypt the target instruction of an illegal control-flow transfer. The state will only be valid if it can be reached following the state computation chain from the start of the target instruction's basic block and related IV. Given that SaeCAS conceals IV and k information, the decryption process is secure as long as the PRINCE function is secure.

C. Analysis of Previous Approaches

We now discuss issues with previous approaches. For our evaluation, we assume that the attacker has no access to the encryption secret being used to encrypt the code. The attacker may attempt to recover this secret by exploiting flaws on the *design* of the security mechanism, but not its *implementation*. We also assume that a cryptographically secure cipher is being utilized to decrypt instructions, and that a full control-flow graph for the application is available.

1) Classic ISR Approaches: As previously stated, classic ISR approaches offer no protection against code-reuse attacks. This stems from the fact that decryption is preformed in a way that resembles electronic code book. In fact, code-reuse attacks were developed as a result of code injection no longer being possible after write-no-execute primitives were introduced.

In some cases, such as with the PRINCE cryptographic module in the NXP LPC55S69, data leakage is still possible using a read anywhere vulnerability targeting the program flash space. Since decryption is transparent when data is read from the microcontroller's flash memory, a load instruction will be able to recover the plaintext.

2) The issue with C^3: Control-flow Carrying Code uses Shamir's Secret Sharing scheme with $n = 3$ of k parts needed to recover the key which properly decrypts the basic block. These three parts are the source address, destination address, and a *master secret* which is used as an extra parameter. However, we now consider the case where two different control-flow instructions target the same unique basic block as in the case with two function calls targeting the same function, or where a single control-flow instruction targets two different basic blocks as in the case of virtual function pointers. We assume that for either case the attacker has used an information disclosure vulnerability to leak the source and destination addresses of the possible control-flow paths. At this point, the attacker has managed to obtain three of the $n = 3$ pieces required to compute the secret a_0 which decrypts the target basic blocks in lieu of having the master secret.

What is more, since the attacker is capable of obtaining the corresponding Lagrange polynomials, the original polynomial can be recovered. Using it, the attacker can determine all addresses which will be decrypted a_0 when reached from one of the leaked sources. Although the instructions being decrypted at these address may produce results which are different from the intended (valid) ones, the decryption process becomes *predictable*. The attacker can utilize these findings to construct a useful gadget catalog and launch a code-reuse attack which will go undetected by C^3. This is to say, *C^3 is incapable of enforcing a control-flow graph in its full extent under an information disclosure vulnerability even if the attacker is unable to obtain the master secret.*

The situation is exacerbated in a microcontroller, where memory management and protection techniques are rarely used. The polynomial gives the attacker the locations where code can be injected encrypted with the secret key a_0. Redirecting execution to that site from the known sources will result in the instruction stream being decrypted properly. With this, the attacker is able able to gain total control of the platform and bypass the CFI policy.

3) Analysis of SCFP: For our analysis of Sponge-Based Control-Flow Protection (SCFP), we will represent updates to the state of the system by application of the operation $S_{n+1} = f(S_n)$. We will also represent the "patch" to the state with the operation $S_j = P(c, S_k)$, where c is the delta in the program stream, S_k is the current state, and S_j is the desired state.

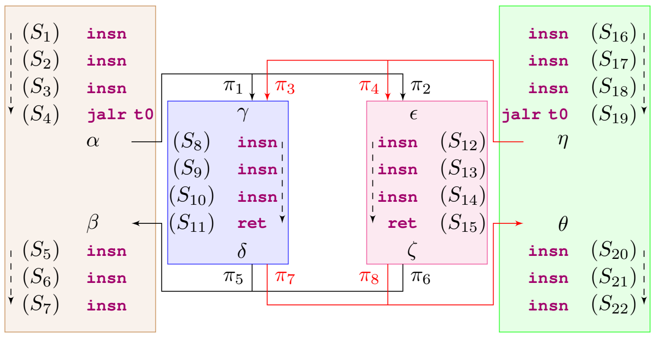
For our analysis, we will use the construct in Figure 5. Figure 5b shows the overall control-flow graph of the scenario. Two functions can make indirect calls to one of two possible callees, then return back to its caller. We show the SCFP-instrumented code in 5a, broken down into the contents of the basic blocks. Each instruction has a unique cipher state associated with them, represented as (S_i). The cipher state is required to properly decrypt the instruction. Symbols α to θ are the deltas or patches which need to be applied to the state of the cipher to properly decrypt target instructions after control-flow transfers π_1 to π_8 have taken place.

We begin our analysis by demonstrating how the deltas are computed. We start with control-flow paths π_1 and π_3. We note that the cipher must be at state S_8 for the first instruction of the basic block to properly decrypt regardless of caller. We apply the operation $S_8 = P(\gamma, P(\alpha, S_4))$ when following transition π_1, and the operation $S_8 = P(\gamma, P(\eta, S_{19}))$ when following transition π_3. Consequently, $P(\gamma, P(\alpha, S_4)) = P(\gamma, P(\eta, S_{19})$ thus $P(\alpha, S_4) = P(\eta, S_{19})$. Similar analysis following transitions π_2 and π_4 yields $P(\alpha, S_4) = P(\eta, S_{19})$.

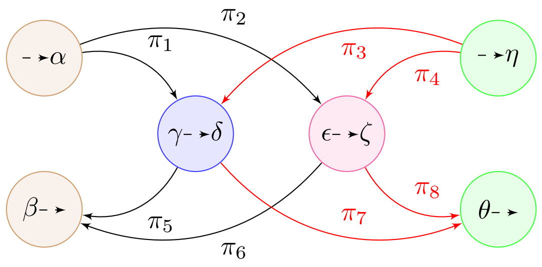
From this, we conclude that *the synchronization of the cipher actually happens with the first patch that is applied in the transition*. The cipher is already synchronized before entering the target basic block. The application of the second patch is so that $S_8 \neq S_{12}$, as to avoid state reuse.

As such, to instrument the code, we can pick any random values for γ, δ, ϵ, and ζ so that $\gamma \neq \epsilon$, and $\delta \neq \zeta$. To compute η we can pick a random α, then $P(\alpha, S_4) = P(\eta, S_{19})$, so that $\eta = P_S^{-1}(S_{19}, P(\alpha, S_4))$. We can compute S_4 and S_{19} by successive applications of f to states S_1 and S_{16}, respectively. That is, $S_4 = f^{(3)}(S_1)$ and $S_{19} = f^{(3)}(S_{16})$. We repeat the same process to compute δ and ζ.

As demonstrated, the synchronization of the cipher at return sites is *not* dependent on β and θ, but on δ and ζ. The cipher



(a) The decryption cipher must be at state S_k in order to properly decrypt the accompanying instruction. Symbols α to θ represent the deltas or patches applied to the cipher on control-flow changes. Solid arrows represent valid control-flow paths, and are numbered π_1 to π_8. Dashed arrows represent sequential execution within a basic block.



(b) Control-flow graph of presented code. Nodes of the same color represent basic blocks of the same function.

Fig. 5: Example of SCFP-instrumented code (top), and its control-flow graph (bottom).

will have the same state regardless of which path is taken from the set $\{\pi_5, \pi_6, \pi_7, \pi_8\}$. *As such, the CFI scheme is unable to check we are returning to the proper caller. We can traverse the path $S_4 \to \pi_1 \to S_8 \to S_{11} \to \pi_7 \to S_{20}$ and still be able to decrypt instructions properly, even though this execution path is illegal. The proper execution path starting at S_4 must end in S_5*. That is to say, the CFI policy can not conform to the stateful requirements presented in [27].

Moreover, assume that there is a new execution path π_9 which targets a valid call site S_{23} from S_{19}. We also assume that this call target can only be accessed through this path. We must then apply patches η and ι to set the cipher at state S_{23}. Then, $S_{23} = P(\iota, P(\eta, S_{19}))$. However, recall from our analysis that when traversing π_4, we have that $S_{12} = P(\epsilon, P(\eta, S_{19}))$. Hence, $P(\eta, S_{19}) = P_p^{-1}(\iota, S_{23}) = P_p^{-1}(\epsilon, S_{12})$. But as previously seen $P(\eta, S_{19}) = P(\alpha, S_4)$. Consequently $P_p^{-1}(\iota, S_{23}) = P(\alpha, S_4)$, which results in $S_{23} = P(\iota, P(\alpha, S_4))$

Therefore, *S_4 becomes an unintended caller of S_{23}, introducing a non-existent path π_{10} into the control-flow graph. Furthermore, when returning from this new callee, S_5 also becomes a valid return site, regardless of the path that was taken to enter the function*. In other words, even if we are able to fully recover the control-flow graph for an application, SCFP is *provably unable* to enforce a fine-grained control-flow policy, and thus is unable to provide the properties of authenticated execution.

D. Comparison of SAECAS and Previous Approaches

As demonstrated, previous authenticated execution approaches fail at providing all the theoretical safeguards of this type of defense. Based on previous sections, we compare the capabilities of previous approaches as shown in Table IV based on how well approaches can enforce control-flow integrity (CFI), encrypted execution (EE), and guarding software intellectual property (IPG).

TABLE IV: Comparison of our approach to previous authenticated execution mechanisms in terms of control-flow integrity protection enforcement (CFI), encrypted execution enforcement (EE), and software intellectual property protection (IPG). Approaches flagged with ●, ◐, or ○ offer strong, weak, or no protection in the category, respectively.

Approach	**CFI**	**EE**	**IPG**
Traditional ISR [12], [13], [14], [15]	○	◐	○
Scylla [9]	◐	●	○
SOFIA [18]	◐	●	○
NXP LPC55S69 PRINCE Module [16], [17]	○	●	◐
Control-Flow Carrying Code [10]	◐	◐	◐
Sponge-Based Control-Flow Protection [11]	◐	●	●
SAECAS (this work)	●	●	●

As discussed in previous sections, traditional instruction set randomization (ISR) approaches serve a functionality equivalent to the write-no-execute (W$\oplus$X) memory protection primitive. Consequently, it is incapable of enforcing any kind of CFI policy. Moreover, binaries reside in plain-text on disk and are only encrypted when loaded to memory. Thus, traditional ISR can not provide IP protection to software. Lastly, the ISR approaches use weak encryption, and the key is known to the OS thus not being able to fully provide encrypted execution. Much of this is applicable to the LPC55S69 PRINCE module, as it does not provide control-flow integrity. Also, its IP software safeguards are not resilient to memory leak attacks. Data reads by software from flash memory are transparently decrypted by the PRINCE module, thus a memory read vulnerability can be employed to obtain the machine code of the software running.

Scylla and SOFIA provide weak control-flow integrity policies as part of their design. We also demonstrated how Control-Flow Carrying Code and Sponge-Based Control-Flow Protection provide equally weak CFI policies. Carlini et al. showed how the enforcement of a weak CFI policy leads to a class of attack called control-flow bending in [39]. In this type of attack an adversary is capable of launching Turing Complete code-reuse attacks by using the presence of extraneous edges in a control-flow graph caused by a weak CFI policy.

Although Scylla and SOFIA provide a strong encrypted execution model, both approaches must ship the decryption key with the software in a way that is accessible to the operating system. As such neither approach can properly safeguard software IP. Furthermore, the key generation scheme used in C^3 gives the attacker the possibility of predictable decryption of software, weakening the encrypted execution model and software IP guard guarantees.

We demonstrated how SAECAS is capable of providing resilient coverage over all three facets of authenticated execution due to its handling of control-flow, and IVs for generating cryptographic keys. The guarantees of SAECAS hold as long as the cryptosystem used is resilient to attacks.

VIII. Conclusion

In this paper we evaluated previous authenticated execution (AE) approaches. We presented a new architecture which is capable of providing strict AE without the need for software instrumentation, which we then evaluated in terms of hardware, and software overhead. We proved the security aspects of our approach while demonstrating that previous approaches will fail at providing the guarantees of AE. As future work, we plan to move our work into an ASIC platform, further optimize our architecture for low-power demands, and add support for multitasking platforms.

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, National Science Foundation (NSF-1801599), and the Semiconductor Research Corporation (2018-TS-2860). 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] D. Goodwin, "Hack causes pacemakers to deliver life-threatening shocks," https://arstechnica.com/information-technology/2018/08/lack-of-encryption-makes-hacks-on-life-saving-pacemakers-shockingly-easy/, 2018.

[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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] O. Arias, D. Sullivan, H. Shan, and J. Jin, "LAHEL: Lightweight Attestation Hardening Embedded Devices using Macrocells," in *2020 Hardware Oriented Security and Trust Symposium (HOST)*. IEEE, 2020.

[9] D. Sullivan, O. Arias, D. Gens, L. Davi, A.-R. Sadeghi, and Y. Jin, "Execution integrity with in-place encryption," 2017.

[10] Y. Lin, X. Cheng, and D. Gao, "Control-flow carrying code," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 3–14.

[11] M. Werner, T. Unterluggauer, D. Schaffenrath, and S. Mangard, "Sponge-based control-flow protection for iot devices," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 214–226.

[12] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 272–280.

[13] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 281–289.

[14] G. Portokalidis and A. D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 41–48.

[15] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis, "Asist: architectural support for instruction set randomization," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 981–992.

[16] NXP Semiconductors, *LPC55S6x 32-bit Arm Cortex-M33 Microcontroller*, 2020, rev. 1.9.

[17] ——, "LPC55S69 Security Solutions for IoT," 2019, rev. 0.

[18] R. de Clercq, J. Gtzfried, D. bler, P. Maene, and I. Verbauwhede, "Sofia: Software and control flow integrity architecture," *Computers & Security*, vol. 68, pp. 16 – 35, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404817300664

[19] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[20] J. He, X. Guo, H. Ma, Y. Liu, Y. Zhao, and Y. Jin, "Runtime trust evaluation and hardware trojan detection using on-chip em sensors," *2020 57th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2020.

[21] J. He, H. Ma, X. Guo, Y. Zhao, and Y. Jin, "Design for EM side-channel security through quantitative assessment of rtl implementations," *25th Asia and South Pacific Design Automation Conference*, 2020.

[22] H. Ma, J. He, Y. Liu, Y. Zhao, and Y. Jin, "Security-driven placement tools for electromagnetic side channel protection," *2019 Asian Hardware Oriented Security and Trust (AsianHOST)*, 2019.

[23] VectorBlox Embedded Supercomputing, "ORCA FPGA Optimized RISC-V," 2016.

[24] Intel, "Avalon interface specifications," 2020, mNL-AVABUSREF 2020.01.03.

[25] ARM, "Amba axi and ace protocol specification documentation," 2019, aRM IHI 0022G (ID073019).

[26] OpenCores, "Wishbone b4: Wishbone system-on-chip (soc) interconnectionarchitecturefor portable ip cores," 2010.

[27] Y. Jin, D. Sullivan, O. Arias, A.-R. Sadeghi, and L. Davi, "Hardware control flow integrity," in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, 2018, pp. 181–210.

[28] C. A. Lattner, "Llvm: An infrastructure for multi-stage optimization," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2002.

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

[30] W. Zhang, Z. Bao, D. Lin, V. Rijmen, B. Yang, and I. Verbauwhede, "Rectangle: a bit-slice lightweight block cipher suitable for multiple platforms," *Science China Information Sciences*, vol. 58, no. 12, pp. 1–15, 2015.

[31] G. Research, "LEON3 synthesizable processor." [Online]. Available: http://www.gaisler.com

[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[33] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger *et al.*, "Prince–a low-latency block cipher for pervasive computing applications," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2012, pp. 208–225.

[34] J. Jean, I. Nikolić, T. Peyrin, L. Wang, and S. Wu, "Security analysis of prince," in *International Workshop on Fast Software Encryption*. Springer, 2013, pp. 92–111.

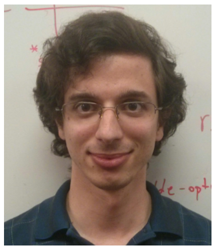
[35] R. Posteuca and G. Negara, "Integral cryptanalysis of round-reduced prince cipher," *Proceedings of the Romanian Academy, Series A*, vol. 16, pp. 265–270, 2015.

[36] H. Jarollahi, V. Gripon, N. Onizawa, and W. J. Gross, "Algorithm and architecture for a low-power content-addressable memory based on sparse clustered networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 4, pp. 642–653, 2014.

[37] M. Zackriya and H. M. Kittur, "Precharge-free, low-power content-addressable memory," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 8, pp. 2614–2621, 2016.

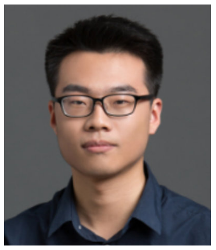
[38] W. Choi, K. Lee, and J. Park, "Low cost ternary content addressable memory using adaptive matchline discharging scheme," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018, pp. 1–4.

[39] 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)*, 2015, pp. 161–176.



Orlando Arias is a Computer Engineering Ph.D. student at the University of Florida. He received his B.S. and M.S. degrees in Computer Engineering from the University of Central Florida. His research interests include device security, secure computer architectures, network security, IP core design and integration, and cryptosystems. Mr. Arias was a recipient of the Best Paper Award in the 52nd Design Automation Conference as part of his work on hardware-assisted control-flow integrity systems, and a recipient of the A. Richard Newton Young Scholar Fellow award in the 52nd and 53rd Design Automation Conference. He was also awarded the NSF GRFP in 2015.

Dean Sullivan is a Ph.D. student in the Department of Electrical and Computer Engineering at the University of Florida (UF). He received his B.S. degree in Electrical Engineering and M.S. degree in Computer Engineering from the University of Central Florida. His research interests include system security and software-based microarchitectural side-channels attacks. Mr. Sullivan was a recipient of the Best Paper Award in the 52nd Design Automation Conference as part of his work on hardware-assisted control-flow integrity, and a recipient of the A. Richard Newton Young Scholar Fellow award in the 52nd and 53rd Design Automation Conference.



Haoqi Shan is a Ph.D student in the Department of Electrical and Computer Engineering at University of Florida. He received his bachelor degree in Electrical Engineering at Harbin Engineering University, China in 2015. His research interests include cyber security and IoT security.

Yier Jin (M'12-SM'19) is an Associate Professor and IoT Term Professor in the Department of Electrical and Computer Engineering (ECE) in the University of Florida (UF). Prior to joining UF, he was an assistant professor in the ECE Department at the University of Central Florida (UCF). He received his PhD degree in Electrical Engineering in 2012 from Yale University after he got the B.S. and M.S. degrees in Electrical Engineering from Zhejiang University, China, in 2005 and 2007, respectively. His research focuses on the areas of hardware security, embedded systems design and security, trusted hardware intellectual property (IP) cores and hardware-software co-design for modern computing systems. Hardware-Oriented Security and Trust in 2017, the 2018 ACM TODAES, the 28th edition of the ACM GLSVLSI in 2018, and the Design, Automation and Test in Europe in 2019. He is also interested in the security analysis on Internet of Things (IoT) and wearable devices with particular emphasis on information integrity and privacy protection in the IoT era. Dr. Jin is a recipient of the DoE Early CAREER Award in 2016 and ONR Young Investigator Award in 2019. He received Best Paper Award at DAC15, ASP-DAC16, HOST17, ACM TODAES18, GLSVLSI18, and DATE'19. He is also the IEEE Council on Electronic Design Automation (CEDA) Distinguished Lecturer.