Abstract. Software components are frequently used in cyber-physical systems (CPSes) to control a physical mechanism, such as a valve or brakes on a car. These systems are extremely sensitive to software vulnerabilities, as their exploitation could lead to injury, damage to equipment, or environmental catastrophe. This paper proposes a hardware-based security architecture called SOFIA, which protects software running on microprocessors used in CPSes. SOFIA provides mechanisms to protect software integrity and control flow integrity. This allow the processor to defend against a large number of attacks, including code injection, code reuse, and fault-based attacks on the program counter. In addition, the architecture also defends against software copyright infringement and reverse engineering. All protection mechanisms are enforced in hardware using cryptographic techniques. We are the first to propose a mechanism to enforce control flow integrity at the finest possible granularity using cryptographic techniques. A SOFIA core has been created by implementing the proposed architectural features on a LEON3 microprocessor. The SOFIA core requires that its software conforms to a strict format. To this end, we additionally designed and implemented a software toolchain to compile source code that adheres to the formatting rules. Several benchmarks were compiled with the SOFIA toolchain, and were executed on a SOFIA core running on an FPGA, showing an average total execution time overhead of 106% compared to an unmodified LEON3 core. Our hardware evaluation shows a clock speed reduction of 23.2%.

Keywords: Control Flow Integrity, Hardware, Security, Computer architecture, Instruction Set Randomization
SOFIA: Software and Control Flow Integrity
Architecture

Ruan de Clercq¹, Johannes Götzfried², David Übler²,
Pieter Maene¹, and Ingrid Verbauwhede¹

¹ KU Leuven, Belgium - ESAT/COSIC and imec
² Department of Computer Science, FAU Erlangen-Nuremberg

1 Introduction

Cyber-physical systems (CPSes) enable the physical world to integrate with control systems such as embedded devices or the Internet of Things (IOT). Software algorithms run on embedded devices which use sensors to measure physical processes and actuators to control physical components, such as a valve or the brakes on a car. The software is responsible for monitoring and controlling these physical components to ensure that they are operating correctly. Examples of CPSes include industrial control systems, process control, autonomous automobile systems, and medical implants.

The correct functioning of CPSes is crucial, as the failure can lead to injury, damage to equipment, or environmental catastrophe. To ensure their correct operation, we need to ensure that the software that runs on the computational components are not compromised. Exploits that target vulnerabilities in the underlying software are increasingly used to obtain full system access. The attack surface is also increasing, as processors become more interconnected through ad-hoc networks and through the public Internet. Despite a significant amount of research to address the underlying problems of software vulnerabilities, there are still a vast number of attacks that threaten the security of software.

This work aims to protect the software running on low-end microprocessors used in CPSes. We specifically target software applications that do not require an operating system. Low-end microprocessors often lack basic architectural support for security, and are frequently deployed in the field, where it is easy to extract and exploit their software. As we rely on software to control and monitor physical processes, we need to know that the software behaves in a predictable manner, and an adversary should not be able to alter their software or tamper with their operation. Ideally, even if an attacker obtains the code running on a device, he should not be able to understand it and know, e.g., which version of the software is being deployed. Not knowing that will make it harder to exploit potential weaknesses in the software, such as overflows or incomplete input validation.

A lot of research has focused on code injection [43,46] and control flow integrity (CFI) [3,4], but current solutions [9,17,45,50,53,54,55] have been demonstrated to be breakable [13,20,27,48], or they require significant hardware [5,15,18,19,33,34,40] or OS support found only on high end general-purpose processors. CFI
forces a program to follow a control flow path along a Control Flow Graph (CFG) that can be predetermined or calculated at run-time.

**Our contribution.** In this paper we propose SOFIA, an architecture that defends against attacks based on code injection, code reuse, software tampering, and fault attacks on control flow. The architecture is deeply integrated in a processor's pipeline stages, and relies on cryptographic methods to protect the running software. In particular, SOFIA enforces control flow integrity, software integrity, and code secrecy. In detail, our contributions are as follows:

- We offer reliable tampered code protection, i.e., tampered instructions or instructions that occur during illegal control flow will not be executed. Furthermore, metadata stored in memory is protected from tampering.
- We protect against powerful adversaries in control of all memory as well as against fault attacks on control flow.
- Our policy is enforced entirely in hardware and does not enforce protection by means of any software stored in memory.

In addition we provide a thorough, real-world evaluation of SOFIA. To this end, we designed and implemented a compiler-based toolchain to transform software to be executed on a SOFIA core. We further implemented a SOFIA core in VHDL as an extension to the LEON3 soft processor. In addition, the performance overhead was evaluated by executing a number of different benchmarks on the SOFIA core programmed on an FPGA.

The remainder of this paper is structured as follows. First, the problem statement is provided including the threat model and system goals. Next, the proposed architecture is described. Afterwards, we present a description of the hardware implementation, followed by implementation details of the toolchain. We then describe the current state-of-the-art in control flow integrity and give a security and performance evaluation of our solution. Finally, we conclude our work with an outlook over future research directions.

## 2 Problem Statement

In this section we discuss the system model, composed of a threat model and a set of system requirements.

### 2.1 Threat Model

This work considers an adversary capable of the following:

- Full control of program and data memory. The attacker can replace parts or all of the software, including the entire stack, program memory, data memory, ROM and RAM.
- Full control of external I/O pins of the processor. He is able to probe the external memory bus and disconnect external components. However, the internal components of the processor cannot be altered or probed.
• Capable of performing non-invasive fault attacks that target the program flow, such as glitching the clock. However, other types of fault attacks that do not target the program flow, such as glitching the ALU or bus of the processor, are not considered part of the attacker model.

• Side-channel attacks are not considered.

• With respect to cryptographic capabilities, we follow the Dolev-Yao [22] model, where an attacker is capable of performing protocol-level attacks, but cannot break cryptographic primitives.

2.2 System goals

In this work, a hardware-based security architecture performs run-time verification of the integrity and execution of software. To this end, the requirements of the system are as follows.

Software integrity: The attacker should not be able to execute tampered software on a SOFIA core. We consider the software to consist of instructions and read-only data.

Control flow integrity: The attacker should not be able to change the control flow of running software along an invalid path without this being detected. This includes software-based attacks based on code-reuse such as jump-oriented-programming (JOP) [10], return oriented programming (ROP) [49] and return-to-libc [51], and further includes hardware-based control flow attacks [52], such as instruction skips induced by glitching the clock.

Tampered code protection: No tampered code should be allowed to execute on the processor. We consider tampered code to include illegally modified instructions or any code resulting from an invalid control flow.

Code confidentiality: The attacker cannot read stored software in clear or execute stored software on other devices. This prevents the attacker from finding potential vulnerabilities and further prevents the extraction of confidential IP from the software.

Reverse engineering protection: The software that is stored on each device cannot be copied and executed on other devices.

3 Architecture

In this paper, we propose a number of architectural extensions to a microprocessor in order to enhance its security. The extensions consist of two major mechanisms. First, a Control Flow Integrity (CFI) mechanism guards against code injection and code reuse attacks. This mechanism uses a type of Instruction Set Randomization [35] where each instruction is encrypted with control flow dependent information. At runtime, the instructions are decrypted using the same control flow dependent information. Only when the correct control flow paths are taken the instructions will decrypt correctly.

Second, a Software Integrity (SI) mechanism ensures that tampered software never executes on the processor. Here, a Message Authentication Code (MAC) is
used to verify the integrity of groups of instructions at run-time. If an integrity violation is detected, the processor is reset in order to prevent any tampered instructions from executing.

An overview of the architecture is shown in Fig. 1. Encrypted instructions \( (c_{\text{inst}}) \) are fetched from program memory, placed in instruction cache, and decrypted by the CFI feature. The decrypted instructions \( (\text{inst}'_{\text{i}}) \) are sent to the Instruction Decode (ID) stage of the processor. At the same time, the SI feature performs run-time integrity verification of the decrypted instructions. Upon detection of an integrity violation, execution is halted by resetting the processor, thereby preventing both tampered control flow and tampered instructions from executing. The processor should be able to reboot reliably fast, allowing the software to quickly reach a safe and expected state. Each processor is embedded with a set of unique keys that can only be accessed by the block cipher. The keys are known only by the device manufacturer and the software provider.

In the remainder of this section, the CFI and SI mechanisms will first be presented as standalone features. Afterwards, there will be a discussion on how to use the two mechanisms within a single system. As shown in Table 1, each standalone feature meets only part of the criteria of the system model. However, when the standalone features are combined into a single system (CFI and SI), they complement each other, thereby satisfying all the criteria in the system model.

### 3.1 Control Flow Integrity (CFI)

The main idea of the CFI mechanism is to perform ISR by decrypting instruction opcodes based on control flow dependent information. This enforces CFI by ensuring that instructions are only decrypted correctly at runtime when valid control flow paths are followed. A binary that consists of encrypted instructions is created by performing a transformation operation at compile time. The instructions are encrypted based on the control flow paths present in a precise Control Flow Graph (CFG) of the whole program. Each encrypted instruction is decrypted at run-time using a combination of the current program counter and the address of the previously executed instruction.

Each instruction in the binary is encrypted using a block cipher in counter (CTR) mode, as shown in Alg. 1. The counter value \( (I_{\text{i}}) \) contains the dynamic control flow between two instructions. This is expressed as the address of the currently executing instruction together with the address of the previously executed instruction. Encryption is performed with \( c_{\text{inst}} = E_{k_1}(I_{\text{i}}) \oplus \text{inst}_{\text{i}} \), while decryption is performed with \( \text{inst}'_{\text{i}} = E_{k_1}(I_{\text{i}}) \oplus c_{\text{inst}} \), with \( I_{\text{i}} \) the counter value and \( k_1 \) the encryption key. The counter is \( I_{\text{i}} = \{\omega \parallel \text{prevPC}_{\text{i}} \parallel \text{PC}_{\text{i}}\} \), with \( \text{PC} \) the current program counter or address of \( \text{inst}_{\text{i}} \), \( \text{prevPC} \) the previously executed program counter, and \( \omega \) a nonce. The nonce \( \omega \) needs to be unique across each version of every encrypted program, and is stored in a fixed address in the binary. This decryption process is illustrated in Fig. 2.

Instructions are decrypted correctly as long as the control flow of a running program follows the paths of the original CFG. However, when a program is
**Algorithm 1**: Control flow dependent information is used to encrypt and decrypt the instructions of a program.

**Input**: Plaintext $p_i$, $j$-bit key $k_i$, number of plaintext blocks $u$, nonce $\omega$

**Result**: Encryption produces $r$-bit ciphertext blocks $c_0, \ldots, c_u$. Decryption recovers plaintext $m$.

**Encryption**:

for $i \leftarrow 1$ to $u$
do

$I_i = \{\omega \parallel \text{prevPC}_i \parallel \text{PC}_i\}$;

$O_i \leftarrow E_{k_i}(I_i)$;

$t_i \leftarrow$ the $r$ least-significant bits of $O_i$;

$c_i \leftarrow m_i \oplus t_i$;

end

**Decryption**:

for $i \leftarrow 1$ to $u$
do

$m_i \leftarrow c_i \oplus E_{k_i}(I_i)$, where $I_i$, $O_i$, and $t_i$ are computed as above.

An attacker typically has to force control to flow along a path which does not exist in the original CFG, e.g., to execute injected code or to perform a code reuse attack. This will cause at least one instruction to be decrypted incorrectly, as the counter $I_i$ contains an invalid $\text{prevPC}$.

An example program listing with corresponding CFG is shown in Fig. 3. Each CFG node represents a single encrypted instruction, while the edges indicate control flow between instructions. The solid edges represent valid control flow, with the encryption counter $I_i$ indicated next to each edge. The CFG shows that control flows from node 1 to 2; therefore, instruction 2 is decrypted with counter value $I_2 = \{\omega \parallel 1 \parallel 2\}$. A branch causes control to flow from node 2 to 5; therefore, instruction 5 is decrypted with counter value $I_5 = \{\omega \parallel 2 \parallel 5\}$. When an attacker causes invalid control flow to occur from, e.g., node 1 to node 5, instruction 5 is decrypted with counter $I_5 = \{\omega \parallel 1 \parallel 5\}$, leading to a decryption error.

Function calls are supported in a similar way as direct branches. The function’s entry point is encrypted with the call site, while the return point in the call site is encrypted with the address of the return instruction in the callee. Callees with multiple call sites or the targets of function pointers with multiple call sites correspond to nodes with multiple predecessors in the CFG, and cannot be handled with the scheme discussed so far. Section 3.4 discusses the necessary extensions.

The CFI mechanism presented in this section provides protection from attacks based on code injection and code reuse. However, a decryption error caused by tampered control flow might lead to a decrypted instruction (inst') with a valid opcode. The instruction will execute on the processor, albeit leading to a different result than that of the original program. This is a serious problem, as the incorrectly decrypted instruction could lead to a malicious result. This problem can be solved by using the CFI mechanism in combination with the SI mechanism described in the following section.
3.2 Software Integrity (SI)

This section presents a mechanism which ensures, with very high probability, that only untampered instructions can execute on the processor. A Message Authentication Code (MAC) is precomputed on groups of instructions, and stored in instruction memory, as shown in Fig. 4. At run-time, a MAC verification is performed on each group of instructions before they reach the end of the processor’s pipeline. The run-time MAC is compared with the precomputed MAC to verify the integrity of all instructions in each group. If their verification fails, the processor is reset in order to prevent tampered instructions from executing.

Design An execution block, shown in Fig. 5, consists of $m$ MAC words ($M_i$) and $n$ instructions ($inst_i$). Control can only flow into an execution block at $M_1$, and can only exit at $inst_n$. Inside the execution block, control flow passes through each MAC word and then through each instruction.

The processor’s Instruction Fetch (IF) pipeline stage is used to read instructions and precomputed MAC words from memory. The MAC words are replaced with a `nop` before being sent to the decode stage. It is necessary that all words in an execution block are fetched every time it is executed, as all the instructions in a block are needed to compute the run-time MAC, and the precomputed MAC is required for verification.

In our design we use the Cipher Block Chaining-Message Authentication Code (CBC-MAC) algorithm [32] with a 64-bit MAC length. In the remainder of the text we will refer to the two 32-bit MAC words as $M_1$ and $M_2$. It is well known that the CBC-MAC algorithm is only secure for messages of a fixed length [29]. Care needs to be taken, as SOFIA computes a MAC on different message lengths due to the two block types that each consists of a different number of instructions (see Section 3.5). We propose to address this issue by using a different key for each type of block, thereby using one key for each message length. We further use a different key for the MAC and for encryption. Consequently, each device has a total of three different keys: $k_1$ is used for encryption, $k_2$ is used for CBC-MAC of execution blocks, and $k_3$ is used for CBC-MAC of multiplexer blocks.

Preventing tampered blocks from executing One of the design criteria is to prevent the execution of instructions that are tampered with or occur after an illegal control flow. Here we discuss the techniques used to achieve this.

SOFIA is designed to work as an extension to any microprocessor. However, for the discussion in this paper, we will base the design on the instruction pipeline of the SPARCv8-based LEON3 processor as an example. It uses a single-issue pipeline with seven stages:

1. Instruction Fetch (IF) requests instructions from cache or main memory.
2. Instruction Decode (ID) translates opcodes into instructions and generates call or branch target addresses.
3. Operand Fetch (OF) reads the operands from registers.
4. Execute (EXE) performs ALU, logical, and shift operations. For memory operations, the address is generated.
5. **Memory Access (MA)** performs read or write operations to/from memory.
6. **Exception (XCP)** resolves traps and interrupts.
7. **Write Back (WB)** stores the result of the ALU, logical, shift, or memory access operations to the register file.

A CPS’s physical components commonly interface with a microcontroller through a physical connection, such as General Purpose Input/Output (GPIO) pins. The microcontroller’s software controls the actuators by writing to the interface’s memory mapped addresses with *store instructions*. Special care needs to be taken with store instructions, as they could be used to send tampered commands to an actuator, which could have a catastrophic effect, e.g., disable the brakes on a car, or by emptying a patient’s insulin tank.

SOFIA detects tampered instructions by verifying an execution block’s integrity while the instructions are partially executed inside the processor’s instruction pipeline. In this work we propose to verify the integrity of a block before any store instructions inside a given block has reached the Memory Access (MA) pipeline stage. Instructions other than stores cannot control actuators from CPSes, and are therefore allowed to progress through all the processor’s pipeline stages. A simple approach, illustrated in Fig. 6, is to make the execution blocks small enough to fit into the pipeline stages before the MA stage. This allows the run-time MAC to be computed before the instructions reach the MA stage. If verification fails, the instructions are prevented from moving further in the pipeline by resetting the processor, thereby discarding all instructions in the block before reaching the MA stage.

In the LEON3, *memory access* operations are performed in the fifth instruction pipeline stage. Therefore, a four-instruction execution block can fit in the pipeline stages before the MA stage. When a single-cycle MAC hardware component is used, tampering can be detected before the first instruction reaches the MA stage. To improve the performance of the system, the number of instructions in an execution block can be increased to six instructions if store instructions are not allowed to be located on \( \text{inst}_1 \) or \( \text{inst}_2 \), as illustrated in Fig. 7. A *store violation* is generated when a store instruction is detected on \( \text{inst}_1 \) or \( \text{inst}_2 \). Store violations are handled in the same manner as MAC violations.

### 3.3 Control Flow Integrity with Software Integrity (CFI and SI)

By using both the CFI and SI mechanisms in a single processor, it is possible to detect tampered execution blocks as well as invalid control flow. By designing the system to rapidly detect tampering we are able to also prevent the execution of instructions resulting from tampered control flow. The CFI mechanism decrypts instructions based on the run-time control flow, but can not detect decryption errors. The SI mechanism performs integrity verification in order to detect tampered instructions, but cannot detect invalid control flow when used alone. Therefore, to detect tampered control flow and instructions, the SI mechanism verifies the integrity of a block only after the CFI mechanism has decrypted the encrypted instructions. Fig. 8 shows the process of decrypting the words in an execution block using CTR mode, and then calculating the CBC-MAC on all
the decrypted instructions (\(\text{inst}_i^{\text{d}}\)). The figure assumes untampered control flow, i.e., each counter value \(I_i\) contains the valid values for \(\text{PC}_i\) and \(\text{prevPC}_i\), such that \(\text{PC}_i = \text{addr}(\text{inst}_i)\), and \(\text{prevPC}_i = \text{addr}(\text{inst}_{i-1})\) or \(\text{prevPC}_i = \text{callAddr}\), with \(\text{callAddr}\) the address of the call site.

At run-time the CFI mechanism first decrypts the instructions using dynamic control flow information. Next, the SI mechanism calculates the run-time MAC on the decrypted instructions. If an invalid control flow path was taken, a decryption error occurs. When the SI mechanism calculates the run-time MAC with the incorrectly decrypted instruction and incorrect MAC is produced and the integrity verification fails, the processor is then reset to prevent the execution of instructions resulting from the tampered control flow.

The plaintext binary is transformed with the MAC-then-Encrypt construction \(^{[42]}\). For each execution block, a MAC \(M\) is first calculated on the plaintext instructions. Next, \(M\) is interleaved with the instructions to form execution blocks. Finally, the plaintext execution blocks are encrypted with Alg. \(^{[1]}\) and then stored in main memory.

### 3.4 Blocks with Multiple Predecessors

The CFI mechanism presented in Section 3.1 only supports nodes with a single predecessor, since execution blocks only have one entry point. This section introduces the multiplexer block which allows for two predecessors. This block uses both the CFI (Section 3.1) and SI (Section 3.2) mechanisms.

Just like for the execution block, a two-word MAC \(M\) is first calculated on the block’s plaintext instructions \(\text{inst}_i\). To support two predecessors, we propose to make two entry points by inserting two copies of the first MAC word \(M_1\) at the beginning of the block, as shown in Fig. 9. Each copy of \(M_1\) is used as an entry point into the block, which we call \(M_{1e1}\) and \(M_{1e2}\). Each of the two entry points are therefore encrypted using their respective call sites (\(\text{callAddr}_{1}\) and \(\text{callAddr}_{2}\)), as illustrated by Fig. 10. The two entry points are encrypted as follows: \(c_{M_{1e1}} = E_{k_1}(I_1) \oplus M_1, I_1 = \{\omega \parallel \text{callAddr}_{1} \parallel \text{addr}(c_{\text{inst}})\}\), and \(c_{M_{1e2}} = E_{k_1}(I_2) \oplus M_1, I_2 = \{\omega \parallel \text{callAddr}_{2} \parallel \text{addr}(c_{\text{inst}})\}\). In addition, two distinct control flow paths exist in the block. The first control flow path enters the multiplexer block at \(c_{M_{1e1}}\), skips \(c_{M_{1e2}}\), and flows to \(c_{M_2}\), followed by all the encrypted instructions \(c_{\text{inst}}\) in the block. The second control flow path enters the block at \(c_{M_{1e2}}\), flows to \(c_{M_2}\), followed by all the encrypted instructions \(c_{\text{inst}}\) in the block. To decrypt \(c_{M_2}\), the counter value \(I_3 = \{\omega \parallel \text{addr}(c_{\text{inst}}) \parallel \text{PC}\}\) is used by the hardware, regardless of which control flow path is used.

If a node in the CFG requires more than two predecessors a tree of multiplexer blocks can be used. In Fig. 11 a multiplexer tree allows a node to be called by four different call sites. The tree structure is used to handle entry points from call sites, function pointers, and branch targets. Therefore, the multiplexer tree structure needs to have an entry point for each call site that can reach a function through a branch or a function call.
3.5 Support for blocks with single and multiple predecessors

Most non-trivial programs consist of blocks with one entry point as well as blocks with multiple entry points. In the above text we outlined two different types of blocks, namely the execution block with a single entry point, and the multiplexer block which has two entry points. In order to create a meaningful program using these two blocks, we need to develop mechanisms to make them work together within the same system.

The software needs a mechanism to indicate to the hardware which type of block is about to execute. We propose to solve this by using the call site to inform the hardware of the block type. For an execution block, we select the block’s first word $c_{M_1}$ as the call site. Therefore all calls, branches, or fall-throughs to $c_{M_1}$ will indicate to the hardware that an execution block should be executed. For a multiplexer block we propose to use the second and third words, respectively $c_{M_{e2}}$ and $c_{M_2}$, as the two call sites. Therefore, a branch or a call to $c_{M_{e2}}$ or $c_{M_2}$ will indicate to the hardware that a multiplexer block should be executed. A branch/call to $c_{M_{e2}}$ will cause the first control flow path to be followed, and similarly, a branch/call to $c_{M_2}$ will cause the second control flow path to be followed.

The size of both block types is chosen to be eight 32-bit words. Therefore, the execution block consists of 2 MAC words and 6 instructions, while a multiplexer block consists of 3 MAC words and 5 instructions.

3.6 MAC Chaining

This section discusses the security of decrypting parts of a MAC with different CTR-mode operations, and proposes some changes to improve its security. The discussion here is limited to execution blocks, but the proposed modifications are also applicable to multiplexer blocks.

When using the architecture presented so far, the first encrypted MAC word ($c_{M_1}$) is decrypted with a counter value ($I_1$) that depends on the call site ($callAddr$), while the second MAC word is decrypted with a counter value ($I_2$) that depends on $addr(c_{M_1})$. This reduces the security of the CFI mechanism to only 32-bits as only the first MAC word is decrypted using $callAddr$. During CTR-mode decryption we ideally want all MAC words to be decrypted using a counter value that depends on $callAddr$. However, currently the system will only provide 32-bit security for control flow as only the first MAC word is decrypted using a counter consisting of $callAddr$.

To solve this problem, we propose to use additional cryptographic operations to chain all the MAC words together, thereby ensuring that the decryption of the entire MAC relies on $callAddr$. We propose two different solutions. For $size(M) \leq b$, a single block cipher operation can be used to perform the chaining, and an ECB-mode encryption can be used. However, for $b > M$, multiple cryptographic operations are necessary, for which we recommend using CBC-mode encryption to chain the MAC words together. For both cases encrypting the binary works as follows. First, $M$ is encrypted with either ECB or CBC mode, providing
us with $c_{M_1}$ and $c_{M_2}$. Next, $c_{M_1}$ and $c_{M_2}$ are encrypted with CTR-mode, i.e. $c_{c_{M_1}} = E(I_1) \oplus c_{M_1}$, with $I_1 = \{\omega \parallel \text{callAddr} \parallel \text{addr}(c_{c_{M_1}})\}$, and $c_{c_{M_2}} = E(I_2) \oplus c_{M_2}$, with $I_2 = \{\omega \parallel \text{addr}(c_{c_{M_1}}) \parallel \text{addr}(c_{c_{M_2}})\}$. At runtime, the reverse operations are performed. First, CTR-mode decryption will be used: $c'_{M_1} = c_{c_{M_1}} \oplus I_1$, and $c'_{M_2} = c_{c_{M_2}} \oplus I_2$. Next, the partially decrypted MAC words ($c'_{M_1}$ and $c'_{M_2}$) will be de-chained using either ECB-mode or CBC-mode decryption. Using this approach ensures that $M'_2$ can only be decrypted correctly if $M'_1$ was also decrypted correctly. In order for $M_1 = M'_1$, the correct call site (callAddr) is needed. This solves our problem, as the decryption of the entire MAC now relies on callAddr.

**Chaining with ECB-mode:** The plaintext MAC words are encrypted using ECB mode, i.e., $\{c_{M_1} || c_{M_2}\} = E(M_1 || M_2)$. The partially decrypted MAC words are decrypted as follows: $\{M'_1 || M'_2\} = D(c'_{M_1} || c'_{M_2})$.

**Chaining with CBC-mode:** The plaintext MAC words are encrypted using CBC mode, i.e., $c_{M_1} = E(M_1 \oplus IV)$, and $c_{M_2} = E(M_2 \oplus c_{M_1})$. The partially decrypted MAC words are decrypted as follows: $M'_1 = D(c'_{M_1}) \oplus IV$, and $M'_2 = D_k(c'_{M_2}) \oplus M'_1$.

## 4 Hardware implementation

This section describes the SOFIA hardware implementation. First, we provide an overview of the modifications made to the LEON3 processor. Next, we discuss the choice of block cipher, followed by a description of the hardware design. Finally, we discuss the schedule employed by the block cipher.

### 4.1 Overview

SOFIA has been implemented on Gaisler’s LEON3 v1.3.7-b4144 soft microprocessor. The processor was configured with the minimum number of peripherals, branch prediction support, DDR memory support, a small amount of cache, and a single vector trap table. The hardware design was evaluated on a Xilinx Virtex-6 XC6VLX240T FPGA.

The majority of the modifications to the processor were done in the seven-stage integer pipeline (`iu3.vhd`). The CFI component was placed between the LEON3’s cache controller (`ico`) and the ID pipeline stage. In addition, the logic to calculate the next program counter was modified in order to allow for the non-standard control flow through multiplexer blocks. A reset line was wired from `iu3.vhd` to the top level design (`leon3mp.vhd`) in order to halt execution of instructions when either an integrity violation is detected or a store instruction is detected on inst$_1$, . . . , inst$_3$.

### 4.2 Block cipher

In order to prevent tampered instructions from executing, the MAC computation needs to complete in only a few cycles. A simple approach is to use a single cycle block cipher, thereby allowing the decryption of each instruction in only one
cycle. This further allows for rapid MAC verification when using CBC-MAC, as each decrypted instruction \((\text{inst}^i)\) can be processed in a single cycle.

For our initial design we selected the RECTANGLE-80 [56] block cipher, which has a 64-bit block size, and an 80-bit key. The RECTANGLE-80 block cipher was unrolled to compute in one cycle [39]. Our initial experiments showed that placing a single-cycle implementation of RECTANGLE-80 in the instruction pipeline stages of the processor increased the design’s critical path to a maximum clock frequency of around 28 MHz. Block ciphers are typically complex, leading to long critical paths when all of its operations are performed in a single cycle. Therefore, in order to improve the critical path of the SOFIA core, we decided to use a dual cycle cipher. In addition, we made a second implementation which uses a dual cycle version of PRINCE [11], which has a 64-bit block size and a 128-bit key. Table 2 compares the hardware overhead of each block cipher configured for running in either one or two cycles.

### 4.3 Hardware design

A block diagram of the hardware is shown in Fig. 12. The two stages of the dual cycle block cipher are indicated by "Cipher part one" and "Cipher part two", with the pipeline register indicated with \(\text{r\_pipeline}\). As discussed in Section 3, the block cipher is used in three different modes of operation. First, to perform the CTR-mode decryption the cipher uses the input \([\omega \| NPC \| FPC]\), where NPC indicates the address that will be fetched in the next IF slot, and FPC is the address in the current IF slot. The first execution block serves as the starting point of the program, and can be entered from anywhere. To this end, \(I_1 = [\omega \| FPC \| 0]\) is used to decrypt the first MAC pair in the first execution block. For each CTR-mode decryption, the output of the cipher is stored in the \(\text{r\_cipher\_out}\) register. In the following cycle, encrypted instructions arriving from the cache controller (\(\text{ico\_data}\)) are decrypted, stored in the \(\text{r\_decrypted\_inst}\) register, and sent to the processor’s decode stage. In the same cycle, encrypted MACs are decrypted, stored in the \(\text{r\_precomputed\_mac}\) register, and a \(\text{nop}\) is sent to the decode stage.

Second, for the MAC dechaining operation, the CTR-mode decrypted MAC \((C_M')\), which was stored in \(\text{r\_precomputed\_mac}\) in the previous slot, is provided as an input to the cipher. The result of this operation \((M')\) is finally stored inside the \(\text{r\_precomputed\_mac}\) register.

Third, for the CBC-MAC operation, the first cipher input consists of the decrypted instruction stored inside \(\text{r\_decrypted\_inst}\), and the result is stored inside \(\text{r\_runtime\_mac}\). For the second and last CBC-MAC operation, \(\text{r\_decrypted\_inst}\) is XORed with \(\text{r\_runtime\_mac}\) and are provided as an input to the cipher. When the final MAC result has been computed it is immediately compared with \(\text{r\_precomputed\_mac}\).

The reset logic is generated from two different sources. First, the output of the cipher is compared to the value stored in \(\text{r\_precomputed\_mac}\) during the first cycle when the lowest six bits of \(FPC\) are equal to eight, which corresponds to the first cycle in which the MAC computation is finished. Second the first three decrypted
instructions ($\text{inst}_1, \ldots, \text{inst}_4$) are decoded to detect STORE instructions. When
the reset line is asserted the processor immediately resets, causing the annulation
of all partially executed instructions, as well as the cancellation of all pending
memory accesses.

While experimenting with the design, we found a large delay present in
the FPC and NPC signals generated by the LEON3 together with our own
logic which drives the input signals to the first block cipher stage. To allow
the design to run at a high clock frequency, $r_{\text{pipeline}}$ was not placed in
the middle of the block cipher, but was rather placed after only a couple of
rounds of computation. This leads to the critical path of the design being from
$r_{\text{pipeline}}$ to $r_{\text{precomputed\_mac}}$. We further improved the timing of the design
by partitioning the block cipher to a specific region of the FPGA.

### 4.4 Scheduling the Block Cipher

In our implementation, we use a single block cipher instance to perform the
following three different operations. First, CTR mode decryption is performed
on the encrypted instructions ($c_{\text{inst}}$) and encrypted MAC ($C_{\text{MAC}}$). Second, an
ECB decryption operation is performed on $C_M$ to de-chain the MAC, and finally
obtain $M'$. Third, CBC-MAC is used to calculate the runtime MAC on the
decrypted instructions $\text{inst}_i'$. Since our design’s block cipher has 64-bit blocks,
a single operation can process two 32-bit words. This means that a total of
four CTR-mode operations are required to decrypt all the words in a block,
one operation is required for MAC de-chaining, and three CBC operations are
required to calculate the MAC.

A timing diagram of the block cipher operations to process a single execution
block is shown in Fig. 13. Two different instruction pipeline stages of the LEON3
are indicated in the figure, namely the Instruction Decode (ID) slot number,
and the Memory Access (MA) slot number. In this example, the execution block starts
at slot number zero, and ends at slot number seven. The figure shows that $MAC$
verification occurs at MA slot number five. Since a dual cycle block cipher is used,
all crypto operations span two slots. In addition, the block cipher is pipelined
and a new input can be fed to the cipher in every slot.

The four CTR-mode operations are indicated with $CTR_0, \ldots, CTR_3$. The
result of each CTR operation is used to decrypt an instruction in the ID stage.
When scheduling the CTR operations, it is important to ensure that all instruc-
tions can be decrypted before reaching the ID stage, e.g., the computation of
$E(I_0)$ needs to finish before slot zero reaches the ID stage.

The MAC de-chaining operation is indicated by $ECB$, and finishes in ID slot 2.
The CBC-MAC operations, indicated by CBC-MAC$_0, \ldots,$ CBC-MAC$_2$, are used
to calculate the MAC over the decrypted instructions. The final MAC operation
(CBC-MAC$_2$) starts in ID slot 7, and finishes in slot 8, at which point the MAC
verification is performed. To meet the requirement that tampered instructions
doesn’t execute STORE instructions are disallowed in $\text{inst}_3, \ldots, \text{inst}_3$. 
4.5 Limitations

This section discusses the current limitations of our SOFIA implementation.

Interrupts are currently not supported. However, this is not a fundamental limitation and could be supported with additional hardware logic. One approach to achieve this is to save the internal SOFIA registers on a protected stack before entering an ISR, and then restoring the internal SOFIA registers when returning from an ISR. In addition, the hardware should allow the first block of each ISR to be entered from any predecessor. A simpler approach is to delay executing an ISR until after processing the last instruction in the currently executing block. This has the advantage that the internal SOFIA registers will not need to be saved/restored from the stack.

We acknowledge that side-channel attacks are a threat to the security of our system. If SOFIA would be deployed in a real-world system, then the keys would need to be protected against side channel attacks. Fortunately, these attacks can be mitigated by using countermeasures, such as masking [14].

5 Software Implementation

The SOFIA hardware extension imposes several constraints on software which should run on the modified processor. In particular, there are constraints regarding the control flow between blocks as well as the type and position of instructions within blocks. To be able to compile and run standard C code, we have designed a software toolchain consisting of several parts, pre-linkage as well as post-linkage. Because the complete control flow graph must be known to produce SOFIA-compatible machine code, all source files need to be passed to the toolchain for compilation. Our toolchain is able to compile ANSI C down to an ELF binary satisfying all constraints with just the limitations given by SOFIA itself.

5.1 Toolchain Design

Our toolchain consists of several independent tools. We added optimization passes and changed the SPARC backend within the LLVM compiler infrastructure. An unmodified version of clang is used to compile source code to LLVM intermediate code. The intermediate representations for all source files are then linked together into a single file using llvmlink which is then passed to opt. The optimizer applies two custom optimization passes to the program to ensure a binary control flow graph within each function, i.e., at most two predecessors are allowed per node, and a binary call graph between functions, i.e., each function is only allowed to have two direct predecessors, before passing the result to llc. For llc we changed the SPARC backend in such a way that it emits SPARC assembler instructions respecting the constraints of SOFIA blocks instead of plain assembler instructions.

Within the second stage, the binutils provided by the Bare C Compiler (BCC) from Aeroflex Gaisler are used to assemble and link the code emitted by
Furthermore, unnecessary sections such as comment or debug sections are stripped from the resulting binary using `objcopy`.

For the final stage, the binary is processed by custom standalone tools written in C++. These tools encrypt the SOFIA blocks within the binary and assist the programmer by verifying that the final binaries comply with the SOFIA constraints. An overview of how the independent parts of the toolchain work together is shown in Figure 14. Each step will be described further in the following sections.

**Compiler Stage** The LLVM Compiler Infrastructure is used to transform ANSI C to assembly code as shown in Figure 15. The tools used to achieve this are `clang`, `llvm-link`, `opt`, and `llc`. Clang is the compiler frontend provided by the LLVM project. It takes C source code as input and emits LLVM intermediate code. No changes were necessary for the purpose of this project.

The intermediate code generated by `clang` is then linked using `llvm-link`. This step is needed, because building a binary call graph between functions requires global knowledge of the program. To ensure that each function is only called by two other functions, each function needs access to its callers.

Two optimization passes were written to ensure a binary control flow graph, i.e., each node in the graph is only allowed to have at most two predecessors. Each pass implemented the same algorithm (see Section 5.2) but operates at different scopes, namely intra- and inter-function. At this point, a single LLVM intermediate code file is produced. The last step of the compiler stage uses `llc` to compile the intermediate code to assembly code.

As SOFIA is currently not able to handle traps, but register window overflows and underflows would cause traps, the SPARC register window was disabled. To this end, we implemented the `flat` calling convention as the first backend patch for LLVM, which has the same effect as the `-mflat` option passed to older GCC versions. This leads to an increase of approximately 10% in both code size and execution time.

The second patch for the SPARC backend was written to satisfy the requirements imposed by SOFIA. It consists of two LLVM machine function passes. One transforms the program into block form and marks the beginning of each block. The other ensures that multiplexer blocks can only be reached by explicit branching instructions, but never by falling through.

**Assembler and Linker Stage** BCC is used to assemble the output of the compiler stage and link the resulting object files to a single elf binary. The assembly file is passed to `as`, producing an object file. This object file is then stripped of its comment, note, GNU-stack and eh_frame sections using `objcopy`. The resulting object file is linked with the run-time environment using `ld`.

The run-time environment is implemented partly in C and assembler. The low-level assembler parts are responsible for clearing the BSS segment, providing access to the debug console, and passing control to the main routine. We wrote a small tool which transforms assembler code to SOFIA block form such that the
startup code does not need to be transformed manually. This is necessary, because LLVM provides no frontend for SPARC assembler and thus, our optimization passes cannot be used for low-level assembler routines. The tool is written in C++ and provides similar functionality compared to the optimization passes for LLVM. In particular, a binary control flow graph, block form, and header markers are applied to human readable assembly code. The final result, which is still readable assembly code, is then assembled and linked together with the actual program code.

After this stage, the binary is in block form with marked headers and has a binary control flow graph. All non-cryptographic constraints are satisfied, except the exact offsets for jumps to multiplexer blocks. The adjustment of these offsets, MAC calculation, and encryption of the actual blocks is done after linking, because then all locations are resolved and the offsets necessary for encryption can be obtained.

**Post-linkage Stage** The post-linkage stage has been implemented as a number of standalone C++ applications as shown in Figure [16](#). Those applications are responsible for encrypting the binary, applying the jump offsets to multiplexer blocks, and providing verification routines.

The linked ELF binary from the assembler stage is first passed to find blocks which finds all possible control flow paths and identifies reachable blocks. The list of reachable blocks is later needed by the encryption tool, as the cipher is parametrized with the current and previous program counter. The control flow graph is generated statically by examining the jump targets at the end of each block.

Secondly, with mark blocks, jumps to multiplexer blocks are adjusted to jump to the correct instruction, i.e., an offset is added. Furthermore, blocks are marked as execution or multiplexer block and all necessary information for encryption is prepared and added to the header of each block.

Finally, the encryption tool encrypt encrypts all blocks, with either RECTANGLE-80 or PRINCE, and replaces the block headers with MACs. It outputs the final encrypted ELF binary, but also supports the output of a plain version which contains all instructions in clear and places *nop* s instead of the MAC words within the header. The plain version runs on an unmodified LEON3 processor to simplify debugging.

In addition to the plain version, verification tools are provided which statically check the constraints imposed by SOFIA. The non-cryptographic constraints are verified declaratively in several python scripts, while the correctness of the encrypted data and MAC words are verified by a simple simulator implemented in C++.

### 5.2 Toolchain Implementation

The different stages of our toolchain produce separate result files in such a way that the following stage uses the output file of the previous stage as input. To
simplify the overall compilation process, *CMake* is used to connect the tools and resolve dependencies. In this section we will describe some implementation aspects of our software toolchain in more detail. The constraints for SOFIA are satisfied at different levels. The binary control flow graph, binary call graph, and a single return per function are satisfied with the help of optimization passes, while the fixed block length, single branch instruction per block, position of branching instructions, the position of store instructions are satisfied by backend patches. All remaining constraints, such as encryption and MACs are then satisfied during the post-linkage stage.

**Optimization Passes** Optimization passes in LLVM are shared objects that are dynamically loaded and executed by *opt*. They exclusively work on LLVM intermediate representation and can perform any transformations that does not depend on a particular target machine. Using optimization passes has the advantage that the compiler does not need to be patched and that they can easily be loaded on demand by the optimizer. Furthermore, optimization passes are the best place to implement control flow-related transformations, as LLVM provides an interface to access the different nodes within the control flow graph, and it is possible to rearrange or replace nodes at this level. We used two optimization passes to implement the transformation ensuring a binary control flow graph, i.e., at most two predecessors are allowed per node, within functions and between functions, respectively. This transformation is implemented using an iterative algorithm which operates locally on a node inside a tree and creates proxies to bundle predecessors, until there are at most two predecessors left.

An example run of our algorithm is shown in Figure 17. The current node is shown in blue and it has four predecessors before the transformation (I). In the first iteration (II) the first two predecessors, i.e. 1 and 2, are removed and a proxy node (orange) is created. The current node now still has three predecessors, thus in the second iteration (III) the newly created proxy node and the third original predecessor 3 are chosen and another final proxy node is created for those nodes.

Currently, the choice of the nodes to be bundled in the proxy is arbitrary, as long as they are distinct. However, this choice determines the shape of the resulting tree and provides future optimization opportunities regarding the overall run-time performance.

**Backend Changes** The LLVM intermediate representation offers a convenient way of describing transformations on basic blocks and functions. However, it is agnostic to the target processor’s instruction set and can therefore not be used to satisfy low-level constraints such as the exact position or type of a single assembler instruction. To gain control of architecture-specific instructions, some constraints were implemented in the SPARC backend of LLVM. The backend has access to the SPARC instruction set and offers a lower level view. LLVM intermediate functions are transformed to machine functions and intermediate basic blocks are converted to machine basic blocks. However, the drawback of
implementing backend changes is that they cannot be implemented as separate passes, but instead the compiler itself needs to be patched.

Two machine function passes were written to satisfy the low-level instruction constraints of SOFIA. First, the multiplexer fall-through pass ensures that no multiplexer block is reached without an explicit jump. Next, the basic block inflator lays out the code in SOFIA block form.

The transformation carried out by the basic block inflator is strictly local to basic blocks. It takes a sequence of SPARC instructions and inserts \texttt{nop}s between them until the sequence fits the desired form. The algorithm used to reach this form consists of two phases. In the first phase, all instructions are scanned and an abstract record of blocks is built. This record, named \texttt{SofiaBlockSequence}, consists of a sequence of structures called \texttt{SofiaBlocks}. Each SofiaBlock holds a part of the input instruction sequence, the block type (execution or multiplexer), and the padding required between the instructions.

The algorithm walks through the sequence of instructions and tries to insert them into the current SofiaBlock. The insert operation takes into account the padding required to move the instruction forward and if it has to be at or beyond a specific position inside a block. All instructions that alter control flow, like calls, \texttt{rets} and branches, must be moved to position 7. The instructions that write to memory are moved beyond position 4. If the instruction cannot be placed in the current block, the block is inserted into the SofiaBlockSequence and a new block is created. This may happen either because the block is completely filled or because the required position is already occupied. In any case, each block which is inserted into the SofiaBlockSequence is padded to a fixed size of eight instructions.

This first phase is complete when all instructions are placed in the SofiaBlockSequence. At this point all headers and the necessary padding before and after instructions are known. The second phase walks over all instructions again and inserts padding and header markers into the actual instruction sequence. The header markers are \texttt{unimp} instructions and are always the first instruction of a block. Execution blocks are marked with \texttt{unimp 0x50F1A}, while multiplexer blocks are marked with \texttt{unimp 0x50F1B}. Padding is realized by inserting \texttt{nop} instructions. An example of how the basic block inflator works is shown in Figure 18. It is assumed that the first store instruction is reachable from two blocks, i.e., the first SOFIA block has to be a multiplexer block.

**Cryptographic Operations** The last stage of the toolchain is implemented as several standalone tools written in C++. The tools take the compiled and linked binary emitted by Gaislers bare C compiler as input. This binary is then encrypted as required by the current implementation of the SOFIA processor. The user can choose between the ciphers RECTANGLE-80 and PRINCE to generate the MAC and encrypt the block. In addition to satisfying the cryptographic constraints, jumps to multiplexer blocks are adjusted and the markers introduced by our backend machine passes are replaced by the correct MAC words. This step
has been postponed until after linking, because the offset calculation requires absolute jump targets.

5.3 Limitations

This section discusses the limitations of the current implementation of the SOFIA toolchain.

SOFIA relies on a precise CFG to perform the necessary software transformations. Polymorphism and calculated jumps are currently not supported because they make it difficult to build a precise CFG. The problem of building a CFG from calculated jumps is often approached by over-approximation [36], which leads to losing some security guarantees since an attacker might take paths which do not exist within the real program. Thus, if we would have tool support for creating a precise CFG which allows the jump targets of each calculated jump to be known, it would be possible to support both calculated jumps and polymorphism.

SPARC register windows are currently not supported because they trigger a window overflow or underflow trap when the current window pointer coincides with an invalid window. Therefore, in order to be able to support register windows, the SOFIA hardware needs to be updated to support interrupts (see Section 4.5).

In our evaluation we only considered baremetal applications. However, this is not a fundamental limitation. To provide microkernel support would be challenging, since this kernel type consists of several separate processes, with each running in its own address space. SOFIA requires its software to be inside a single address space, since a single transformation needs to be performed for each process’s software. This is especially problematic for shared libraries, which have a unique address space inside each used process. This can be solved (in part) by using a monolithic kernel, which uses a single large process with one address space. User-mode applications can be supported by disabling shared libraries. To enable task scheduling, interrupt support can be added (see Section 4.5).

Our toolchain currently does not support software-based floating point operations, as well as most optimization options. This can be fixed by implementing optimization passes which replace floating point operations by calls to library functions and ensure compatibility to existing optimization passes.

6 Evaluation

In this section we evaluate the security of the architecture, followed by an evaluation of hardware and performance overheads of our implementation.

6.1 Security Evaluation

SI The SI property is considered equivalent to forging a MAC. An attack is successful if an adversary alters an instruction and MAC pair so that the integrity verification succeeds.
The bit length of a MAC is directly related to the number of trials that need to be performed before a forged message and MAC pair is accepted. For an $n$-bit MAC, an adversary has to perform an average of $2^{n-1}$ random online MAC verifications before this strategy will succeed [29]. Consider that a 64-bit MAC is used, and that an attacker requires at least 8 cycles to verify a forging attempt of a single execution block on the target platform. Then, a successful forgery will require at least 33,001 years to succeed on a 70.6 MHz SOFIA core.

**CFI** The CFI property is considered equivalent to the SI property together with the block cipher’s confidentiality property. An attack is successful if an adversary successfully deviates control flow from the valid CFG.

An attack on the control flow requires two steps. First, the adversary has to divert control flow (e.g., through ROP), from one block to another. Second, the adversary has to forge the MAC and instructions of the second block. Executing the first block will require 8 cycles, while the MAC verification of the tampered block will require an additional 8 cycles, leading to a minimum of 16 cycles per attempt. Therefore, an online brute force attack on a 64-bit MAC will require at least 66,002 years on a 70.6 MHz SOFIA core.

**Tampered code protection** The tampered code protection provided by SOFIA relies on both the CFI and SI properties. At run-time, any code tampering will be detected when executing an execution/multiplexer block. Not only will the tampering be detected, but the detection will happen before write operations occur. This allows the system to ensure that a tampered block, or a block resulting from tampered control flow, will not have a chance to execute a bad instruction that can lead to a tampered write operation. It is especially important to protect write operations in the context of CPS where physical components interface with the processor via a port or memory mapped interface.

**Code confidentiality** The CFI mechanism decrypts instructions at run-time using a key that is unique key to each device. This key is only known by the device manufacturer and by the software provider. Since the software is encrypted for a single device, it can therefore only be decrypted by that device. This prevents an attacker from copying the encrypted software and running it on another device. It is also not possible for an attacker to extract plaintext instructions from a device, as the software is stored encrypted in the cache. By ensuring that code remains confidential, it prevents the reverse engineering of the software to find potential exploits or to obtain a vendor’s software IP.

**Fault attack protection** In SOFIA, instructions and MAC tags are stored encrypted in cache and main memory, and are only decrypted before they are requested by the IF pipeline stage. This allows SOFIA to detect tampered instructions/MAC tags due to fault attacks on the main memory, the instruction cache, or due to control flow glitches, as explained below.
A well-known fault-based attack on control flow is to glitch the external clock line \cite{8}. This typically involves temporarily reducing the clock period, thereby causing the processor to skip instruction(s). In our implementation the block cipher is in the critical path of the processor. Therefore, when the clock period is reduced, the block cipher will be the first component to fail, as its path is the longest in the entire design, meaning that it requires the most amount of time in each clock cycle to perform a computation. In our implementation of SOFIA the block cipher is utilized in all pipeline slots. This means that when the clock period is reduced by a sufficiently large amount (i.e. by glitching the clock), the block cipher will not have had enough time to finish the computation. When the clock is glitched for as little as one clock cycle, the cipher’s operation will be incorrect, leading to a MAC failure. The reason for this is that the MAC can only be calculated if all the cipher operations of a block is computed correctly.

Another well-known fault-attack mechanism is to glitch the power line of a CPU \cite{7}. A deviation of more than 10% of the external power supply could cause problems with an IC. This could lead to wrong computation result of the CPU. SOFIA cannot make any guarantees that the partially executed instructions inside the instruction pipeline have been executed correctly. However, since the block cipher is in the critical path of the processor, it is highly likely to be the first computation to fail as the deviation of the power supply is increased. When one of the block cipher operations fail to compute, a MAC failure occurs, causing a processor reset.

6.2 Hardware Evaluation

Table 3 shows the hardware overhead of the two different SOFIA implementations, with each respectively using RECTANGLE or PRINCE, compared to a LEON3 core. We found that the LUTs increased by 12.9\%, while the clock speed reduced by 23.2\% when compared to an unmodified LEON3 core. The clock speed reduction is due to the block cipher being in the critical path of the design.

6.3 Performance Evaluation

To evaluate the performance of our SOFIA implementations we selected the following software benchmarks applicable to small embedded processors:

- **MiBench** \cite{28}:
  - **ADPCM**: This Adaptive Differential Pulse Code Manipulation (ADPCM) implementation converts an audio file of 16-bit PCM samples into 4-bit samples, thereby yielding a 4\times compression rate.
  - **qsort_small**: The quicksort algorithm is used to sort an array of strings into ascending order.
  - **patricia**: Practical Algorithm to Retrieve Information Coded as Alphanumeric (Patricia) is an algorithm used for routing table lookups. The input data consists of a list of IP traffic from a web server.
- **bitcount**: Performs bit manipulation by counting the bits in an array of integers using five different methods.
- **crc32**: A Cyclic Redundancy Check (CRC) is an operation that is commonly used to detect transmission errors.
  - **EEMBC CoreMark** [23]. This benchmark performs list processing (find and sort), matrix manipulation, state machine, and CRC computation.
  - **AES**: This benchmark performs ECB-mode encryptions using AES-128.

All benchmarks were executed baremetal on three different cores: a LEON3 core clocked at 92.3 MHz, a RECTANGLE-80-based SOFIA core clocked at 60 MHz, and a PRINCE-based SOFIA core clocked at 70.6 MHz. The LEON3 code was compiled with LLVM, while SOFIA code was compiled with the SOFIA toolchain, with both using the compiler flags: `-target sparc -m32 -S -emit-llvm -mcpu=v8`. Fig. [19] shows the cycle overhead of executing the benchmarks on the two SOFIA cores compared to the LEON3 processor. For the RECTANGLE-80 implementation we measured an average cycle overhead of 149%, with AES having the smallest overhead of 36%, and CoreMark having the largest overhead of 438%. For the PRINCE-based SOFIA implementation we measured an average cycle overhead of 141%, with AES having the smallest overhead of 44%, and CoreMark having the largest overhead of 373%. Fig. [20] shows the execution time overhead of executing the benchmarks on the two SOFIA cores compared to executing the benchmarks on the stock LEON3 processor. For the RECTANGLE-80 based SOFIA implementation we measured an average total execution time overhead of 106%, with **crc32** having the smallest overhead of 0.53%, and CoreMark having the largest overhead of 726%. For the PRINCE based SOFIA implementation we measured an average total execution time overhead of 137%, with **crc32** having the smallest overhead of 30%, and CoreMark having the largest overhead of 516%. A comparison of the code size is shown in Table 4 and we found that the code increase by an average of 203%.

### 6.4 Practical feasibility in time constrained CPSes

SOFIA’s performance evaluation, reported in Section 6.3, show significant overhead in terms of memory, clock delay, and execution time. CPSes typically make use of real time programs to guarantee a timely response to critical events. Longer execution times increase the difficulty of meeting deadlines. This raises concerns about the practicality of using SOFIA in a time constrained CPS application. First, the clock delay overhead increases execution time for any program running on a SOFIA core. However, we feel that a clock speed reduction of only 23.2% does not significantly limit the practical feasibility of using SOFIA in a time constrained CPS application.

Second, the memory overhead further increases execution time, since more reads are required from main memory. To mitigate this the processor’s cache size can be increased.

Third, the increase in executed instructions, such as **nops** and jumps, increase the overall execution time. However, the effect of **nops** are minimal because
they execute in one cycle, i.e., they do not cause pipeline stalls due to data dependencies or branch mispredictions. Consequently, the increase in executed instructions is dominated by multiplexer blocks. Large multiplexer trees are especially prevalent at the entry points for frequently used functions. These large trees contain long paths requiring several jumps to reach the intended callee. It is therefore desirable to keep the number of multiplexer blocks inside critical software to a minimum. One approach to achieve this is to write code that ensures that critical functions are called by the minimum number of callers. In addition, the software inside a critical function should contain the minimum number of jumps and calls. Another approach is to modify the toolchain to force it to assign short paths for time critical functions. Yet another approach would be to duplicate frequently used critical functions.

## 7 Related work

Most existing **Control Flow Integrity (CFI)** architectures use a shadow stack to defend against ROP attacks, together with an additional countermeasure to defend against JOP. When shadow stacks are used, a \texttt{CALL} instruction places the caller’s address on both the shadow and normal stack. When returning with a \texttt{RET} instruction, the return address is read from both stacks and are then compared. If they do not match, an exception is raised. SOFIA doesn’t use a shadow stack, but instead relies on two fundamental abilities to enforce control flow integrity: (1) only transitions along the CFG are allowed, as enforced by the CFI and SI mechanisms (this is similar to traditional CFI), and (2) the program is stored encrypted, thereby making it impossible to extract the CFG, and further making it difficult to find gadgets and vulnerabilities. Not knowing the CFG makes it extremely difficult for an attacker to force flow along an unwanted, but existing path.

Software-based CFI solutions \cite{3,9,17,45,50,53,54,55} rely on code placed in a binary to perform control flow checks. In contrast, SOFIA’s security policy is enforced by hardware which cannot be manipulated by an attacker in control of memory. Software-based approaches are often course-grained, and therefore can not detect all control flow violations, as demonstrated by recent attacks \cite{13,20,27,48}. SOFIA enforces CFI at the finest possible granularity (which is the flow between two instructions). A recent work \cite{41} enforces CFI by integrity protecting control flow elements such as return addresses, function pointers, and vtable pointers.

Hardware-based CFI solutions \cite{18,19,25,31,33,34,37,40} rely on a hardware monitor to detect CFI violations. Many existing hardware-based solutions rely on storing sensitive metadata in memory, or use software to form part of their trusted computing base (TCB). This approach is problematic when assuming a powerful adversary which controls all memory, as an attacker can manipulate the metadata, or alter the software-based TCB. In addition, shadow call stacks stored in main memory is also an easy target for such an adversary. In contrast, SOFIA stores only cryptographically protected metadata in memory, and doesn’t
rely use any software in its TCB to enforce its security policy. In \cite{6} CFI and SI are performed in hardware to protect against software and physical attacks.

Control-Flow Bending \cite{12} demonstrated that even when a fully-precise static CFI policy is used, the system is still vulnerable to non-trivial attacks. While this type of attack is possible on SOFIA, it would be extremely difficult to implement, as the attacker cannot extract the CFG from the encrypted binary.

To prevent code injection attacks, recent works \cite{15,21,24,38,40,47} perform integrity verification of instructions at run-time. However, it appears that these solutions cannot reliably prevent the execution of all tampered instructions. In contrast, SOFIA reliably prevents all tampered instructions from executing.

Intel’s Control-Flow Enforcement Technology (CET) \cite{31} uses a shadow stack to defend against ROP, and further uses indirect branch tracking (IBT) to protect against JOP. IBT ensures that forward edges, such as calls and jumps, can only target addresses which contain an endbranch instruction. This only provides coarse-grained protection, since any branch instruction can target all addresses containing an endbranch. In contrast, SOFIA enforces fine-grained CFI and assumes an attacker that is in control of memory, and therefore does not rely on software to form part of its TCB.

Instruction Set Randomization (ISR) \cite{35} is a generic defence mechanism against code injection attacks, which relies on storing encoded instructions, and then decoding the instructions at run-time. Most known ISR implementations \cite{16,16,30,30,44,44} assume that instructions incorrectly decoded instructions will lead to an opcode decode failure, thereby causing a trap in the processor. In contrast, SOFIA detects incorrectly decrypted instructions before execution. One software-based ISR approach \cite{30} used AES in ECB mode in a software dynamic translation system. However, this approach seems to allow an attacker to relocate encrypted instructions without leading to decryption errors. In contrast, SOFIA does not allow blocks of code to be relocated to different addresses, as instructions are encrypted using dynamic control flow (prevPC || PC), which ensures that each instruction can only be located at a single address. ASIST \cite{44} decrypts instructions in hardware using a simple XOR cipher. In contrast, SOFIA relies on a 64-bit block cipher that uses 128-bit secret keys, which makes it non-trivial to derive the encryption key or to forge tampered instructions.

8 Conclusion and Future work

This section discusses future possibilities for improving this work, followed by a conclusion.

8.1 Future work

SOFIA suffers from performance overhead due to increased code size, cycle overhead, and clock speed degradation. This is largely due to the strict set of constraints that are imposed on the software, leading to the insertion of a large number of padding instructions and multiplexer trees. In the future, the
toolchain can be optimized to improve the cycle overhead by reducing the number of inserted and executed padding instructions. Large improvements in cycle overhead could probably be obtained by optimizing the length of the paths taken through multiplexer trees. We further plan to further improve the toolchain by adding support for enabling optimization passes. In addition, we plan to improve the hardware implementation by providing support for interrupts, followed by an evaluation with benchmarks running on an operating system. It would further be interesting to harden SOFIA against side-channel attacks.

8.2 Conclusion

In this paper, we demonstrated the practical feasibility of providing protection against code reuse and code injection attacks using a new security architecture called SOFIA. The architecture’s security policies are enforced in hardware and it protects software through cryptographic mechanisms. Specifically, the architecture provides software integrity protection, fine-grained control flow integrity, and software copyright protection. To evaluate the design, we integrated SOFIA with a LEON3 core, and made an FPGA-based hardware implementation. The SOFIA core increased the hardware area of the LEON3 core by 12.99%, and reduced the maximum clock frequency by 23.2%. In addition, a software toolchain was developed to transform software written in C to conform to the constraints imposed by our architecture. We compiled several software benchmarks with our toolchain to evaluate the overhead imposed by the SOFIA architecture. Altogether SOFIA imposes an average cycle overhead of 141%, and an average total execution overhead of 373% when compared to a stock LEON3 core.

Acknowledgements

We would like to thank Koen de Bosschere, Ronald de Keulenaer, Bjorn de Sutter, Bart Coppens, Bart Preneel, and Bohan Yang for their valuable contributions during the initial development of this work. In addition, we would like to thank Felix Freiling, and Tilo Müller for their support. We would further like to thank the anonymous reviewers. This work was supported in part by the Research Council KU Leuven: C16/15/058. In addition, this work is supported in part by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89), the Flemish Government through G.0130.13N and FWO G.0876.14N, and Cathedral ERC Advanced Grant 695305. Pieter Maene is supported by a doctoral grant of the Research Foundation - Flanders (FWO).

References


43. One, A.: Smashing the stack for fun and profit. Phrack magazine 7(49), 14–16 (1996)


List of Figures
Fig. 1. Overview of the design using CFI and ISR

![Diagram of program memory, cache, CFI, instruction decode, SI, reset, and processor]

Fig. 2. Encrypted instructions ($c_{inst}$) are decrypted at runtime using dynamic control flow information consisting of the current and previous program counters (PC, and prevPC). Under the condition that control flow is untampered, PC = addr($c_{inst}$), and prevPC = addr($c_{inst}$), or prevPC = callAddr, with callAddr the call site.

1: mov r0, r1
2: jmp 5
5: mov r1, r2

$I_1 = \{\omega \parallel 1 \parallel 2\}
I_2 = \{\omega \parallel 2 \parallel 5\}
\omega \parallel 5() = \text{addr}()$

Fig. 3. A CFG of a small program shows two different control flow paths from node 1 to node 5. If the valid control flow path is taken, all instructions are decrypted correctly. However, when the invalid control flow path is taken, instruction 5 is decrypted incorrectly.
Fig. 4. The integrity of a program’s instructions is verified at runtime by comparing the precomputed MAC with the run-time calculated MAC. If verification fails, the processor is reset to prevent tampered instructions from executing.

Fig. 5. The execution block consists of an \( m \)-word precomputed MAC (M) and \( n \) instructions. Control flow can only enter at \( M \), and can only exit at \( \text{inst}_n \). Inside a block the control flows through each consecutive word.

Fig. 6. The instructions in a four instruction execution block fit in the pipeline stages before the Memory Access (MA) stage. This allows the architecture to verify the integrity of the block before a memory access has been performed.
Fig. 7. The size of an execution block can be increased to six instructions if store instructions are restricted from inst1 and inst2.

Fig. 8. The CFI and SI architectural features are combined to detect tampered software and control flow. At runtime, the encrypted words in an execution block (cinsti and CMI) are first decrypted with counter mode, and then a CBC-MAC is used to compute a MAC on the decrypted instructions (inst'i).
Fig. 9. The plaintext multiplexer block uses two copies of the first MAC word $M_1$ as its two entry points, which are respectively called $M_{1e1}$ and $M_{1e2}$.

Fig. 10. The encrypted multiplexer block supports two entry points and has two unique control flow paths through the block.

Fig. 14. Overview of how the independent parts of the SOFIA toolchain work together.
Fig. 11. A tree of multiplexer nodes is used to increase the number of call sites ($C_i$) that can invoke a function.

Fig. 12. A hardware block diagram showing the SOFIA core integrated in the instruction pipeline stages of the LEON3.
<table>
<thead>
<tr>
<th>ID Slot Number</th>
<th>MA Slot Number</th>
<th>MAC Verify</th>
<th>CBC-MAC₂</th>
<th>CTR₁</th>
<th>CTR₂</th>
<th>ECB</th>
<th>CBC-MAC₀</th>
<th>CTR₃</th>
<th>CBC-MAC₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 13. A timing diagram of the block cipher operations to process a single execution block. CTRₙ indicates CTR-mode decryption, ECB indicates MAC de-chaining, and CBC-MAC indicates part of the CBC-MAC computation. The execution block exists in slots zero to seven. Negative slot numbers indicate the previous block in the instruction pipeline. Gray blocks indicate cipher operations of the previous or next block.

Fig. 15. Compiler Stage of the toolchain transforming ANSI C code to SPARC assembler code.
Fig. 16. The post-linkage part of the toolchain is responsible for identifying blocks, adjusting offsets, and finally encrypting each reachable block.

Fig. 17. Example of the iterative transformation ensuring a binary control flow graph. Proxy nodes are added until every node has at most two predecessors.
Fig. 18. Example of how the SOFIA basic block inflator transforms a sequence of assembler instructions to satisfy all low-level constraints.
Fig. 19. A comparison of the cycle overhead of benchmarks running on a SOFIA core compared to a stock LEON3 processor clocked at 92.3 MHz.

Fig. 20. A comparison of the total execution time overhead of benchmarks running on a SOFIA core compared to a stock LEON3 processor clocked at 92.3 MHz.
List of Tables
**Table 1.** Architectural features vs. system model criteria.

<table>
<thead>
<tr>
<th>System goal</th>
<th>CFI</th>
<th>SI</th>
<th>CFI and SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Integrity</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Control Flow Integrity</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Tampered code protection</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Code secrecy</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Tampered control flow prevention</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 2. A comparison of the hardware overhead of two block ciphers: RECTANGLE and PRINCE.

<table>
<thead>
<tr>
<th>Design</th>
<th>Cycles</th>
<th>Slices</th>
<th>LUTs</th>
<th>Flip Flops</th>
<th>Critical path (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECTANGLE</td>
<td>1</td>
<td>699</td>
<td>2,013</td>
<td>0</td>
<td>22.77</td>
</tr>
<tr>
<td>PRINCE</td>
<td>1</td>
<td>348</td>
<td>1,148</td>
<td>0</td>
<td>15.08</td>
</tr>
<tr>
<td>RECTANGLE</td>
<td>2</td>
<td>839</td>
<td>2,016</td>
<td>64</td>
<td>11.4</td>
</tr>
<tr>
<td>PRINCE</td>
<td>2</td>
<td>376</td>
<td>1,154</td>
<td>64</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3. A hardware comparison of SOFIA and a LEON3.

<table>
<thead>
<tr>
<th>Design</th>
<th>Slices</th>
<th>LUTs</th>
<th>Flip Flops</th>
<th>Critical path (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3</td>
<td>6,052</td>
<td>14,222</td>
<td>11,135</td>
<td>92.3 MHz</td>
</tr>
<tr>
<td>SOFIA w/ RECTANGLE</td>
<td>7,208</td>
<td>15,909</td>
<td>11,714</td>
<td>60 MHz</td>
</tr>
<tr>
<td>SOFIA w/ PRINCE</td>
<td>6,728</td>
<td>15,180</td>
<td>11,602</td>
<td>70.6 MHz</td>
</tr>
</tbody>
</table>

Table 4. A comparison of the code size of the benchmarks compiled for both a SOFIA core and for a stock LEON3 processor.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LEON3 code (bytes)</th>
<th>SOFIA code (bytes)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADPCM</td>
<td>4,080</td>
<td>12,480</td>
<td>205</td>
</tr>
<tr>
<td>AES</td>
<td>7,184</td>
<td>38,624</td>
<td>437</td>
</tr>
<tr>
<td>Bitcount</td>
<td>12,272</td>
<td>25,834</td>
<td>110</td>
</tr>
<tr>
<td>CoreMark</td>
<td>22,576</td>
<td>68,640</td>
<td>204</td>
</tr>
<tr>
<td>CRC32</td>
<td>8,736</td>
<td>23,488</td>
<td>169</td>
</tr>
<tr>
<td>Patricia</td>
<td>8,496</td>
<td>23,136</td>
<td>172</td>
</tr>
<tr>
<td>qsort_small</td>
<td>10,896</td>
<td>23,420</td>
<td>123</td>
</tr>
</tbody>
</table>