SOFIA: Software and Control Flow Integrity Architecture

Ruan de Clercq*, Ronald De Keulenaer†, Bart Coppens†, Bohan Yang*, Pieter Maene*,
Koen de Bosschere†, Bart Preneel*, Bjorn de Sutter†, Ingrid Verbauwhede*,
*KU Leuven, Belgium - ESAT/COSIC and iMinds
†Ghent University, Belgium - Computer Systems Lab

***Abstract*—Microprocessors used in safety-critical systems are extremely sensitive to software vulnerabilities, as their failure can lead to injury, damage to equipment, or environmental catastrophe. This paper proposes a hardware-based security architecture for microprocessors used in safety-critical systems. The proposed architecture provides protection against code injection and code reuse attacks. It has mechanisms to protect software integrity, perform control flow integrity, prevent execution of tampered code, and enforce copyright protection. We are the first to propose a mechanism to enforce control flow integrity at the finest possible granularity. The proposed architectural features were added to the LEON3 open source soft microprocessor, and were evaluated on an FPGA running a software benchmark. The results show that the hardware area is 28.2% larger and the clock is 84.6% slower, while the software benchmark has a cycle overhead of 13.7% and a total execution time overhead of 110% when compared to an unmodified processor.**

I. Introduction

Safety-critical systems are used in a large number of applications, including industrial control systems, automotive control systems, and medical implants. The failure or malfunction of these systems can lead to injury, damage to equipment, or environmental catastrophe. These systems commonly include software that runs on a microcontroller. Seeing as software exploits on such systems can have a detrimental effect, it is important to ensure that attackers cannot exploit their software. However, in the past, the security of safety-critical systems has seen little research interest.

This work aims to protect the software running on low-end microprocessors used in safety-critical systems. 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 these processors for safety, they need to behave 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.

Code reuse attacks rely on redirecting control flow through existing code with a malicious result, e.g., jump-oriented programming (JOP) [1] and return-oriented programming (ROP) [2]. These attacks can be mitigated with Instruction Location Randomization [3] or can be prevented by enforcing a **Control Flow Integrity** (CFI) [4] policy. Software-based CFI solutions [4]–[11] are typically course-grained, and therefore can not detect all control flow violations, as demonstrated by recent attacks [12]–[15]. In addition, they rely on software to perform control flow checks, which could be circumvented by a powerful attacker in control of the program memory. Hardware-based CFI solutions [16]–[20] typically use a shadow call stack to mitigate ROP attacks, while using an additional countermeasure to protect against JOP. Most existing hardware-based solutions store sensitive meta-data in data memory, or rely on instructions to form part of their root of trust. The approaches used in [21], [22] offer both CFI and software integrity at run-time, but seem incapable of reliably detecting all tampered instructions.

To prevent code injection attacks, recent works [23]–[27] perform integrity verification of instructions at run-time. However, it appears that no known solution can reliably prevent all tampered instructions from executing.

Instruction Set Randomization (ISR) [28] is a generic defence mechanism against code injection attacks. A software-based approach is followed in [3] where AES is used in ECB mode. However, this approach seems to allow an attacker to relocate encrypted instructions without leading to decryption errors. ASIST [29] decrypts instructions in hardware using a simple XOR cipher, which could make it trivial to derive its encryption key. In [30] a stream cipher is used to encrypt instructions with a seed value that can be updated at run-time.

This paper proposes a new hardware-based security architecture called SOFIA. The architecture adds security features to an existing microprocessor to protect software against attacks based on code injection and code reuse. This creates a system that is exceptionally trustworthy, as the security policy is enforced in hardware, and software copyright and tampering is protected by cryptography. To the best of our knowledge, SOFIA is the first architecture to enforce CFI at the finest possible granularity, and SOFIA prevents the execution of all tampered instructions and instructions resulting from tampered control flow.

The *contributions* of this paper are as follows: (1) a presentation of the architectural modifications needed to provide fine-grained control flow integrity, software integrity, software copyright protection and tampered instruction protection in a single architecture, (2) an evaluation of a hardware implementation on a LEON3 processor, (3) an evaluation of a software benchmark running on the modified processor.

II. Architecture

This paper proposes two mechanisms to enhance the security of a microprocessor. First, a *Control Flow Integrity (CFI)* mechanism guards against code injection and code reuse attacks. This component encrypts each instruction with control flow dependent information. 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.

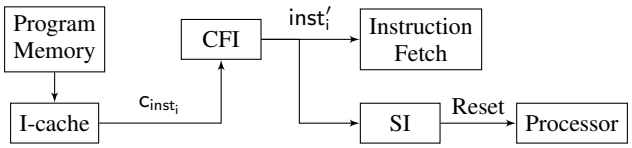

Fig. 1. Encrypted instructions $\mathsf{c}_{\mathsf{inst_i}}$ are decrypted at run-time using control flow dependent information by the CFI component. The SI component verifies the integrity of the decrypted instructions $\mathsf{inst}'_\mathsf{i}$.

The overall architecture is shown in Fig. 1. Encrypted instructions $\mathsf{c}_{\mathsf{inst_i}}$ are fetched from program memory, placed in instruction cache, and decrypted by the CFI feature. The decrypted instructions $\mathsf{inst}'_\mathsf{i}$ are sent to the Instruction Fetch (IF) 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 tampered control flow as well as preventing tampered instructions from executing. The processor should be able to reboot reliably fast, allowing the software to quickly reach a safe and controlled state. Each processor is embedded with a set of unique keys that can only by accessed by the block cipher. These keys are known only by the software provider.

A. 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. 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. The encrypted instructions are decrypted at run-time using a combination of the current program counter and the previously executed program counter.

Each instruction in the binary is encrypted using a block cipher in counter mode, as shown in Alg. 1. The *counter* value is 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 $\mathsf{c}_{\mathsf{inst_i}} = E_{\mathsf{k}_1}(I_i) \oplus \mathsf{inst_i}$, while decryption is performed with $\mathsf{inst}'_\mathsf{i} = E_{\mathsf{k}_1}(I_i) \oplus \mathsf{c}_{\mathsf{inst_i}}$, with I_i the counter value and k_1 the encryption key. The counter is $I_i = \{\omega \,\|\, \mathsf{prevPC}_i \,\|\, \mathsf{PC}_i\}$, with PC the program counter or address of $\mathsf{inst_i}$, prevPC the previously executed program counter, and ω a nonce. The nonce ω needs to be unique across different programs and different program versions of an encrypted program, and is stored in a fixed address in the binary.

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_1, number of plaintext blocks u, nonce ω
Result: Encryption produces r-bit ciphertext blocks c_0 , $\cdots$, c_u.
Decryption recovers plaintext m.

Encryption:
for $i \leftarrow 1$ *to* u **do**
  $I_i = \{\omega \,\|\, \mathsf{prevPC}_i \,\|\, \mathsf{PC}_i\}$
  $O_i \leftarrow E_{\mathsf{k}_1}(I_i)$
  $t_i \leftarrow$ the r least-significant bits of O_i
  $\mathsf{c}_i \leftarrow \mathsf{m}_i \oplus t_i$

Decryption:
for $i \leftarrow 1$ *to* u **do**
  $\mathsf{m}_i \leftarrow \mathsf{c}_i \oplus$, where I_i, O_i, and t_i are computed as above.

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 exploited, 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 causes at least one instruction to be decrypted incorrectly, as the counter I_i contains an invalid previously executed program counter prevPC. The incorrectly decrypted instruction will contain random data.

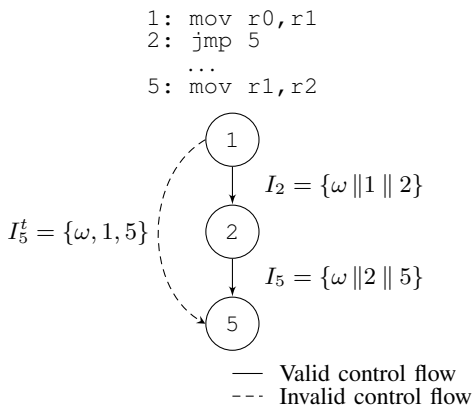

Fig. 2. A CFG of a small program shows two different control flow paths from node 1 to node 5. If the valid control flow path is taken, all instructions are decrypted correctly. However, when the invalid control flow path is taken instruction 5 is decrypted incorrectly.

An example program listing with corresponding CFG is shown in Fig. 2. Each CFG node represents a single encrypted instruction, while the edges indicate control flow between instructions. The solid edges represent valid control flow, with 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 $I_2 = \{\omega \,\|1\,\| 2\}$. A branch causes control to flow from node `2` to `5`; therefore, instruction `5` is decrypted with counter value $I_5 = \{\omega \,\|2\,\| 5\}$. When an attacker causes invalid control flow to occur from, e.g., node `1` to node `5`, instruction `5` is decrypted with counter $I_5^t = \{\omega \,\|1\,\| 5\}$, which leads to a decryption error.

Function calls are supported in a similar way as direct branches. The function's entry point is encrypted with the caller address, while the return point in the caller is encrypted with

the address of the return instruction in the callee. Callees with multiple callers or the call sites of function pointers with multiple callers correspond to nodes with multiple predecessors in the CFG, and cannot be handled with the scheme discussed so far. Section II-D 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 $c'_{\mathsf{inst_i}}$ that has a valid opcode. The instruction will execute on the processor, albeit leading to a different result as 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.

B. Software Integrity (SI)

This section presents a mechanism that 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 is stored in instruction memory, as shown in Fig. 3. At *run-time*, a MAC verification is performed on each group of instructions before they are fully executed through all of the processor's instruction pipeline stages. The run-time MAC is compared with the precomputed MAC to verify the integrity of all instructions in each group. If the verification fails, the processor is reset in order to prevent tampered instructions from executing.

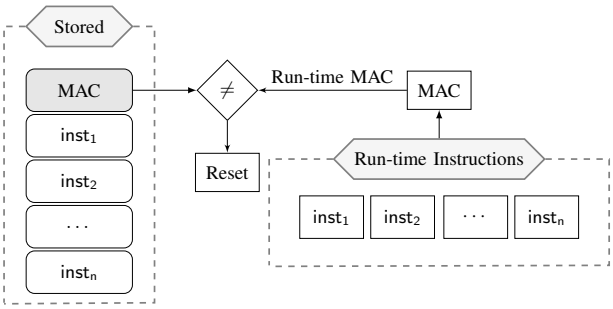

Fig. 3. The integrity of the running program is verified by comparing the precomputed MAC with the run-time calculated MAC. If verification fails, the processor is reset to prevent tampered instructions from executing.

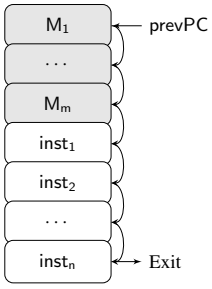

Fig. 4. The *execution block* consists of an m-word precomputed MAC (M) and n instructions. Control flow can only enter at M, and can only exit at $\mathsf{inst_n}$. Inside a block the control flows through each consecutive word.

1) Design: An *execution block*, shown in Fig. 4, consists of m MAC words M_i and n instructions $\mathsf{inst_i}$. Control can only flow into an execution block at M_1, and can only exit at $\mathsf{inst_n}$. 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 [31] 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 [32]. 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 II-E). We propose to fix this problem 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. Therefore, 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 multiplexor blocks.

2) Preventing tampered blocks from executing: SOFIA is designed to work as an extension to any microprocessor. However, in this paper our design is based on the seven stage instruction pipeline of the 32-bit SPARCv8 LEON3 [33] processor.

Store instructions are used for writing to memory and communicating with peripherals via memory mapped and port mapped interfaces. Safety-critical systems often interface with cyber-physical components, which control actuators, such as brakes of a car. In such systems, it is essential that compromised store instructions located in tampered execution blocks are never allowed to execute, as this could have a catastrophic effect, e.g., a store instruction that disables the brakes on a car.

To achieve protection from tampered blocks, we propose that it is sufficient to prevent store instructions in tampered execution blocks from reaching the Memory Access (MA) pipeline stage. To achieve this, the MAC verification is performed before the partially executed instructions of the block reach the MA pipeline stage. A simple approach, illustrated in Fig. 5, is to make the execution blocks small enough to fit into the pipeline stages before the MA stage. The run-time MAC can then 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 preventing all instructions in the block from reaching the MA stage.

When a single-cycle MAC hardware component is used, four instructions in an execution block can fit before the MA stage. However, the number of instructions in an execution block can be increased to six if store instructions are not allowed to be located on inst_1 or inst_2, as illustrated in Fig. 6.

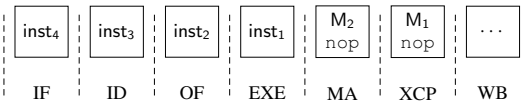

Fig. 5. The instructions in a four instruction execution block fit in the pipeline stages before the Memory Access (MA) stage. This allows the architecture to verify the integrity of the block before a memory access has been performed.

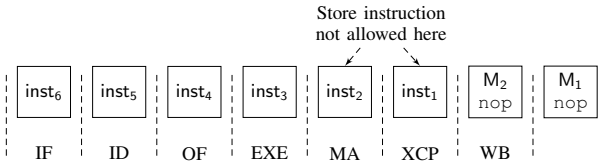

Fig. 6. The size of an execution block can be increased to six instructions if store instructions are restricted from inst_1 and inst_2.

C. Control Flow Integrity with Software Integrity

When the CFI and SI mechanisms are used together they can prevent the execution of instructions resulting from invalid control flow. The CFI mechanism decrypts instructions based on the run-time control flow, but is not capable of detecting 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 invalid control flow, the CFI mechanism first decrypts the instructions, and then the SI mechanism verifies the integrity of a block in order to detect and prevent tampered control flow and tampered instructions.

At run-time the CFI mechanism first decrypts the instructions using control flow dependent information. Next, the SI mechanism calculates the run-time MAC over 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 an 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 [34]. For each execution block, a MAC M is first calculated on the plaintext instructions. Afterwards, M is interleaved with the instructions to form execution blocks, which are then encrypted with Alg. 1.

D. Blocks with Multiple Predecessors

The CFI mechanism presented in Section II-A only supports nodes with a single predecessor, as the execution block only has a single entry point. This section introduces the *multiplexor block* which allows for two predecessors. This block uses both the CFI (Section II-A) and SI (Section II-B) mechanisms.

Just like for the execution block, a two-word MAC M is first calculated on the block's plaintext instructions $\mathsf{inst_i}$. 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. 7. Each copy of M_1 is used as an entry point into the block, which we call $\mathsf{M_{1e1}}$ and $\mathsf{M_{1e2}}$. Each of the two entry points are encrypted using their respective caller addresses prevPC_1 and prevPC_2, as illustrated by Fig. 8. The two entry points are encrypted as follows: $\mathsf{c_{M_{1e1}}} = E_{\mathsf{k_1}}(I_1) \oplus \mathsf{M}_1, I_1 = \{\omega \,\|\, \mathsf{prevPC}_1 \,\|\, \mathsf{PC}\}$, and $\mathsf{c_{M_{1e2}}} = E_{\mathsf{k_1}}(I_2) \oplus \mathsf{M}_1, I_2 = \{\omega \,\|\, \mathsf{prevPC}_2 \,\|\, \mathsf{PC}\}$. In addition, two distinct control flow paths exist in the block. The first control flow path enters the multiplexor block at $\mathsf{c_{M_{1e1}}}$, then skips $\mathsf{c_{M_{1e2}}}$, and flows to $\mathsf{c_{M_2}}$, followed by all the encrypted instructions $\mathsf{c_{inst_i}}$ in the block. The second control flow path enters the block at $\mathsf{c_{M_{1e2}}}$, then flows to $\mathsf{c_{M_2}}$, followed by all the encrypted instructions $\mathsf{c_{inst_i}}$ in the block.

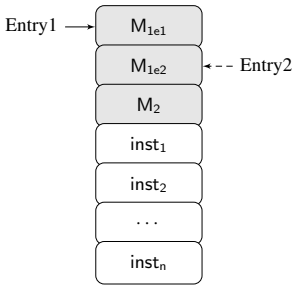

Fig. 7. The plaintext multiplexor block uses two copies of the first MAC word M_1 as its two entry points, which are respectively called $\mathsf{M_{1e1}}$ and $\mathsf{M_{1e2}}$.

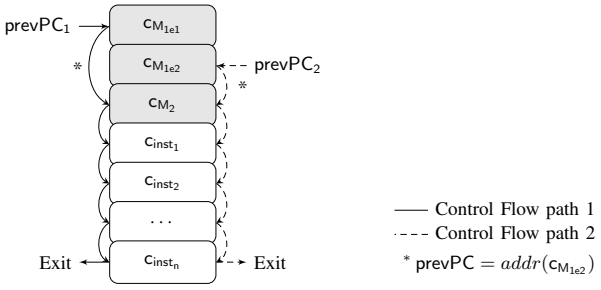

Fig. 8. The encrypted multiplexor block supports two entry points and has two unique control flow paths through the block.

If more than two entry points are required to a single node in a CFG, a tree of multiplexor blocks can be used. Fig. 9 shows how a multiplexor tree allows a node to be called by four different callers. The tree structure is used to handle entry points from call sites, function pointers, and branch targets. Therefore, the multiplexor tree structure needs to have an entry point for each caller that can reach a function through a branch or a function call. This mechanism only works when control flow can be modeled accurately. Therefore, programming language constructs that can lead to control flow that is difficult to analyse, e.g. polymorphism, cannot be addressed by our methods.

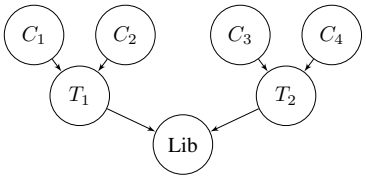

Fig. 9. A tree of multiplexor nodes is used to increase the number of callers (C_i) that can invoke a function.

E. Support for blocks with single and multiple predecessors

Most non-trivial programs consist of blocks with one entry point and blocks with multiple entry points. In the text above we outlined two different types of blocks; namely, the execution block with a single entry point, and the multiplexor 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 inside the same system.

The software needs a mechanism to indicate to the hardware which type of block 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 $\mathsf{c}_{\mathsf{M}_1}$ as the call site. Therefore all calls, branches, or fall-throughs to $\mathsf{c}_{\mathsf{M}_1}$ will indicate to the hardware that an execution block should be executed. For a multiplexor block we propose to use the second and third words, respectively $\mathsf{c}_{\mathsf{M}_{1e2}}$ and $\mathsf{c}_{\mathsf{M}_2}$, as the two call sites. Therefore, a branch or a call to $\mathsf{c}_{\mathsf{M}_{1e2}}$ or $\mathsf{c}_{\mathsf{M}_2}$ will indicate to the hardware that a multiplexor block should be executed. A branch/call to $\mathsf{c}_{\mathsf{M}_{1e2}}$ will cause the first control flow path to be followed, and similarly, a branch/call to $\mathsf{c}_{\mathsf{M}_2}$ 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 multiplexor block consists of 3 MAC words and 5 instructions.

III. Implementation

SOFIA was implemented as an extension to the LEON3 soft microprocessor. The processor was configured with a minimal hardware configuration, and the hardware design was evaluated on a Xilinx Virtex-6 XC6VLX240T FPGA.

The LEON3 was modified to capture all instruction words that pass through the IF pipeline stage. In addition, the logic to calculate the next program counter was modified in order to allow for the complex control flow through multiplexor blocks. Further, a reset line was added from the SOFIA core to the processor in order to halt the execution of instructions when an integrity violation is detected or when a store instruction is detected on inst_1 or inst_2.

To install the software the following approach is followed. First, the source code is compiled into assembly instructions. Next, the assembly instructions are transformed to conform to the format required by the CFI and SI mechanisms. In particular, this means that multiplexor trees are inserted for call sites, function pointer targets and branch targets. Additionally, instructions are transformed into execution blocks and multiplexor blocks. Finally, the assembly code is assembled into machine code and then linked into a binary. For our evaluation the transformed binary was transferred onto the target via the debug interface. However, in production the transformed binary can be stored and executed from the target's non-volatile memory.

For a block cipher we use RECTANGLE-80 [35], which has a 64-bit block size and an 80-bit key. The published version of this cipher requires 26 cycles to perform an operation. In order to prevent tampered instructions from executing, MAC computation needs to occur in only a few cycles. Therefore, the cipher was unrolled to require only two cycles for each operation [36]. This reduces the maximum clock frequency of the processor, as the block cipher increases the critical path of the processor. A single cipher instance is used to perform both the CFI and SI operations. As the cipher has a 64-bit block length, a single operation can process two 32-bit words. Therefore, the cipher alternates between computing CTR-mode and CBC-mode operations every other cycle.

TABLE I
HARDWARE COMPARISON OF SOFIA AND LEON3.

Design	Slices	Clock Speed
Vanilla	5,889	92.3 MHz
SOFIA	7,551	50.1 MHz

IV. Evaluation

A. Security Evaluation

1) 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 [32]. 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. Therefore, a successful forgery of an instruction and MAC pair will require 46,795 years to succeed on a 50 MHz SOFIA core.

2) 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 sucessfully 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). Second, the adversary has to forge the MAC of the first block that is executed after tampering with the control flow. The initial control flow diversion will require 8 cycles, while the MAC verification will require an additional 8 cycles. Therefore, an online brute force attack on a 64-bit MAC will require 93,590 years on a 50 MHz SOFIA core.

B. Hardware Evaluation

Table I shows that the hardware area increased by 28.2%, while the clock speed reduced by 84.6% when compared to an unmodified LEON3 core. The clock speed reduction is due to the block cipher being unrolled 13 times and placed in the critical path of the design.

To benchmark SOFIA we used the MediaBench (I) ADPCM benchmark [37]. It executes bare-metal, and was compiled with the Bare-C Cross-Compiler System for LEON3 from Gaisler. This produced a binary with a text section of 6,976 bytes that executes on an unmodified LEON3 core in 114,188,673 cycles. The transformation process was applied on the compiler-generated assembly, which produced a binary with a text section

of 16,816 bytes. The transformed binary executes on a SOFIA core in 130,840,013 cycles, leading to a cycle overhead of 13.7% and a total execution time overhead of 110%.

V. Conclusion

In this work, we demonstrated that it is practical to provide 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 with cryptographic mechanisms. Specifically, the architecture provides software integrity protection, ultra fine-grained control flow integrity, tampered code protection, 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 28.2%%, and reduced the maximum clock frequency by 84.6%. MediaBench's ADPCM benchmark was executed on the SOFIA core, which shows a cycle overhead of 13.7%, and a total execution overhead of 110% when compared to a stock LEON3 core. Even though the performance overhead is significant, the architecture is still practical for use in safety-critical systems where security and safety are paramount.

An open problem with this architecture is the overhead suffered due to increased code size, execution time, and clock speed degradation. The architecture also does not support virtual memory. In the future we plan to work on design changes to improve the performance of the hardware and perform toolchain optimizations to increase the software performance. We further plan to test the architecture's resistance to fault-based attacks.

Acknowledgments

This work was supported in part by the Research Council KU Leuven: C16/15/058, iMinds, the Flemish Government, FWO G.00130.13N, FWO G.0876.14N, and the Hercules Foundation AKUL/11/19.

References

[1] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *CCS*, 2011.

[2] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS*, 2007.

[3] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill, "Secure and practical defense against code-injection attacks using software dynamic translation," in *Conf. on Virtual Execution Environments*, 2006.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *CCS*, 2005.

[5] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *ACSAC*, 2011.

[6] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones," in *NDSS*, 2012.

[7] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *Conf. on Dependable Systems and Networks*, 2012.

[8] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *USENIX Security*, 2013.

[9] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing." in *USENIX Security*, 2013.

[10] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *IEEE Security & Privacy*, 2013.

[11] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *USENIX Security*, 2014.

[12] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *IEEE Security & Privacy*, 2014.

[13] N. Carlini and D. Wagner, "ROP is Still Dangerous: Breaking Modern Defenses," in *USENIX Security*, 2014.

[14] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX Security*, 2014.

[15] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming," in *IEEE Security & Privacy*, 2015.

[16] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *ISCA*, 2012.

[17] M. Kayaalp, M. Ozsoy, N. A. Ghazaleh, and D. Ponomarev, "Efficiently securing systems from code reuse attacks," *Computers, IEEE Transactions on*, vol. 63, no. 5, 2014.

[18] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek, "Towards a Practical Solution to Detect Code Reuse Attacks on ARM Mobile Devices," in *Workshop on Hardware and Architectural Support for Security and Privacy*, 2015.

[19] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *DAC*, 2014.

[20] L. Davi, Matthias, D. P. Hanreich, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: Hardware-Assisted Flow Integrity Extension," in *DAC*, 2015.

[21] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Secure embedded processing through hardware-assisted run-time monitoring," in *DATE*, 2005.

[22] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," *Computers, IEEE Transactions on*, vol. 59, no. 6, 2010.

[23] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks," in *Scalable Trusted Computing Workshop*, 2009.

[24] A. M. Fiskiran and R. B. Lee, "Runtime execution monitoring (rem) to detect and prevent malicious code execution," in *In'l Conf. on Computer Design*, 2004.

[25] H. Lin, Y. Fei, X. Guan, and Z. J. Shi, "Architectural enhancement and system software support for program code integrity monitoring in application-specific instruction-set processors," *Very Large Scale Integration Systems, IEEE Transactions on*, vol. 18, no. 11, 2010.

[26] R. G. Ragel and S. Parameswaran, "Impres: integrated monitoring for processor reliability and security," in *DAC*, 2006.

[27] J.-L. Danger, S. Guilley, T. Porteboeuf, F. Praden, and M. Timbert, "HCODE: Hardware-Enhanced Real-Time CFI," in *PPREW*, 2014.

[28] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-injection Attacks with Instruction-set Randomization," in *CCS*, 2003.

[29] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis, "ASIST: Architectural Support for Instruction Set Randomization," in *CCS*, 2013.

[30] J.-L. Danger, S. Guilley, and F. Praden, "Hardware-enforced protection against software reverse-engineering based on an instruction set encoding," in *PPREW*, 2014.

[31] International Standard Organization, "Infomation technology - Secruity techniques - Message Authentication Codes (MACs)," ISO/IEC 9797-1:1999(E), 1999.

[32] H. Handschuh and B. Preneel, "Minding your MAC algorithms," *Information Security Bulletin*, vol. 9, no. 6, 2004.

[33] "Cobham Gaisler AB. LEON3 synthesizable processor," http://www.gaisler.com, [Online; accessed 26-Nov-2015].

[34] C. Namprempre, P. Rogaway, and T. Shrimpton, "Reconsidering generic composition," in *Advances in Cryptology–EUROCRYPT 2014*. Springer, 2014.

[35] W. Zhang, Z. Bao, D. Lin, V. Rijmen, B. Yang, and I. Verbauwhede, "RECTANGLE: A Bit-slice Ultra-Lightweight Block Cipher Suitable for Multiple Platforms," *IACR Cryptology ePrint Archive*, 2014.

[36] P. Maene and I. Verbauwhede, "Single-Cycle Implementations of Block Ciphers," in *Lightweight Cryptography for Security and Privacy*, 2015.

[37] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Int'l Symp. on Microarchitecture*, 1997.