2018 IEEE European Symposium on Security and Privacy

Sponge-Based Control-Flow Protection for IoT Devices

Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard
Graz University of Technology
Email: {firstname.lastname}@iaik.tugraz.at

***Abstract*—Embedded devices in the Internet of Things (IoT) face a wide variety of security challenges. For example, software attackers perform code injection and code-reuse attacks on their remote interfaces, and physical access to IoT devices allows to tamper with code in memory, steal confidential Intellectual Property (IP), or mount fault attacks to manipulate a CPU's control flow.**

In this work, we present Sponge-based Control Flow Protection (SCFP). SCFP is a stateful, sponge-based scheme to ensure the confidentiality of software IP and its authentic execution on IoT devices. At compile time, SCFP encrypts and authenticates software with instruction-level granularity. During execution, an SCFP hardware extension between the CPU's fetch and decode stage continuously decrypts and authenticates instructions. Sponge-based authenticated encryption in SCFP yields fine-grained control-flow integrity and thus prevents code-reuse, code-injection, and fault attacks on the code and the control flow. In addition, SCFP withstands any modification of software in memory. For evaluation, we extended a RISC-V core with SCFP and fabricated a real System on Chip (SoC). The average overhead in code size and execution time of SCFP on this design is 19.8 % and 9.1 %, respectively, and thus meets the requirements of embedded IoT devices.

***Index Terms*—control-flow protection, fault attacks, countermeasures, authenticated encryption, sponges**

1. Introduction

Internet-of-Things (IoT) devices serve a variety of purposes, ranging from consumer products in smart home environments, over sensor nodes in modern cars, to control units in critical infrastructures. Typically, these embedded IoT devices feature simple hard- and software architectures with only little consideration of security to stay lightweight. However, the rapidly growing number of IoT devices makes them an interesting target for attackers. In particular, the security of pervasive IoT devices can have direct impact on security and safety in the real world. For example, the worm Stuxnet [25] spread across programmable logic controllers in Iranian infrastructure in 2010, and the malware Industroyer [6] caused a black out of the Ukrainian power grid in 2015. In addition, extensive Distributed Denial-of-Service (DDoS) attacks on infrastructure providers by hijacking a large set of IoT devices, as with the Mirai malware [1], are a significant threat.

There are numerous security challenges with IoT devices. The prevalent Internet connection of IoT devices gives rise to remote attacks on their exposed interfaces. In particular, attackers can try to find and exploit vulnerabilities in these interfaces to take control over IoT devices via code injection or code-reuse attacks, like Return-Oriented Programming (ROP) [39] and Jump-Oriented Programming (JOP) [15]. Further, many IoT devices are run in hostile environments, where attackers have physical access to one or many IoT devices. These physical attackers can read and tamper with code in memory to perform code analysis and inject malicious code. Access to code amplifies the risk of widespread code injection and reuse attacks, and is a critical concern for software Intellectual Property (IP) vendors. However, physical access also allows attackers to perform fault attacks on the processor chip itself, by using, e.g., clock or power glitches [9], [27], [31], in order to manipulate the execution of code on the device. For example, by skipping instructions using power glitches, attackers can get control over one particular IoT device to fake, e.g., sensitive sensor data.

To prevent code injection and code-reuse attacks, different countermeasures have been proposed, such as Data Execution Prevention (DEP), return stack protection [19], [26], [32], Address-Space Layout Randomization (ASLR) [40], software diversification [41] and Control-Flow Integrity (CFI) [2]. To protect the authenticity and confidentiality of IP, encryption and authentication of software binaries and Random Access Memory (RAM) can be used. To counteract physical fault attacks on the control flow of the processor, CFI [43] is a feasible countermeasure as well.

However, current embedded devices hardly implement any of these countermeasures. Moreover, existing countermeasures work well for their original purpose in isolation, but for each of them, some of the attacks on IoT devices remain feasible due to the vast amount of different attack vectors. While a simple combination of existing countermeasures can inhere overheads that are impractical for lightweight embedded devices, the security analysis of combinations of countermeasures can also become highly complex. Recently, SOFIA [20], [21] was presented as the first approach to counteract a combination of these attacks. By encrypting, authenticating and chaining blocks of instructions using a stream cipher and MAC, SOFIA yields CFI as well as confidentiality and authenticity of software. However, one drawback of SOFIA is its checking mechanism. In particular, the dedicated MAC verification is a single point of failure that can potentially be exploited using physical fault attacks

DOI 10.1109/EuroSP.2018.00023

on the error signal within the hardware. Furthermore, the introduced code size and runtime overheads are potentially too high for certain applications.

Contribution. As an alternative approach to SOFIA and to overcome existing limitations, this work presents Sponge-based Control-Flow Protection (SCFP). SCFP is a novel, stateful scheme to protect the confidentiality of software IP and the authenticity of its execution in IoT devices. In particular, SCFP encrypts and authenticates software binaries with instruction-level granularity by using cryptographic sponges. SCFP is designed as a hardware extension that continuously decrypts and authenticates instructions in hardware at the latest possible point before the processor's decode stage.

The use of sponge-based authenticated encryption in SCFP yields fine-grained control-flow integrity and thus prevents code-reuse attacks. By keeping the software encrypted throughout all memory, SCFP completely thwarts code-injection attacks from within software, and effectively protects the IP of software vendors. By decrypting instructions right before the decode stage of the processor, SCFP resists tampering with code in memory, physical attacks on memory like rowhammer [29], [30], and fault attacks that manipulate control flow or software code. SCFP supports interrupt handling and is thus compatible with operating systems. Compared to existing work, SCFP has lower memory and runtime overhead and offers strong fault resistance. In particular, any globally induced physical fault on the processor chip destroys the internal SCFP state with high probability and leads to the execution of random instructions, whereas state-of-the-art CFI schemes use a single verification step that can be by-passed using controlled faults. Random code execution is a secure processor state, because it is hard to control and exploit for an attacker, and has a low probability of being meaningful. Furthermore, timely detection of random execution is also supported.

SCFP is a highly flexible tool. We hence present two suitable sponge constructions as well as three different SCFP instances for different applications. First, Authentic Encrypted Execution (AEE) provides all security features at cryptographic levels of security, i.e., above 80 bits. Second, Authentic Encrypted Execution Light (AEE-Light) reduces memory overhead in trade for reduced software authenticity by using keyed permutations. Third, Infective Execution (IE) is a very lightweight CFI scheme to solely protect against code-reuse and physical fault attacks on the control flow.

SCFP is both practical and lightweight. For demonstration, we integrated AEE-Light with a RISC-V core and evaluated a set of benchmarks on this processor by executing them both unprotected and encrypted with AEE-Light. It shows that the average overheads in code size and execution time of our AEE-Light instance are 19.8 % and 9.1 %, respectively, and thus practical for many IoT scenarios.

Outline. This paper is organized as follows. Section 2 describes the concept of SCFP and the application of authenticated encryption to the instruction stream. Section 3 gives two sponge modes suitable for SCFP and Section 4 presents different SCFP instances and their security properties. Section 5 gives evaluation results and Section 6 provides a comparison with related work. Finally, Section 7 concludes this work.

2. Overall Concept

Sponge-based Control-Flow Protection (SCFP) is a novel security concept for IoT devices that is based on authenticated encryption from cryptographic sponges. In this section, we introduce the threat model we assume for IoT devices and present the architecture of SCFP. In particular, we describe how sponge-based authenticated encryption is applied to an instruction stream and discuss the adaptions required to support arbitrary code execution including control-flow transfers and interrupts.

2.1. Threat Model and Assumptions

This work considers IoT devices which are threatened by both software and physical attacks. In terms of software vulnerabilities, we assume a remote attacker who has arbitrary read and write access to the memory due to bugs in the software. Correspondingly, active physical attackers are assumed to have direct access to the device. This direct access can be used to dump and manipulate external memory, to probe and force signals on the PCB (e.g., bus signals between chips), or to inject global faults into the system (e.g., clock glitches). On the other hand, micro probing and similar invasive techniques are considered out of scope in this work. Similarly, side-channel leakage of hard- and software implementations is not considered in this work.

Presumed targets for adversaries in this domain are to extract secret IP (e.g., firmware code), to bypass security checks (e.g., by skipping one or more instructions), or to achieve arbitrary code execution via code reuse or injection. In other words, adversaries try to compromise the confidentiality and/or authenticity of the code, either at rest or at runtime. Note however, that Denial-of-Service (DoS) as well as data-driven attacks are out of scope given that neither can be solved via a CFI scheme.

This work assumes that SCFP is deployed as the only countermeasure to the mentioned threats. Hence, if guarantees that exceed the capabilities of precisely enforced CFI (e.g., resistance against control-flow bending [17]) are required, additional attack mitigation techniques (e.g., safe stack [32]) have to be utilized. Further, note that the hardware component of SCFP is implemented in such a way that there is no interface to access plaintext instructions, the sponge state or internal SCFP signals. All this information is inaccessible in software.

2.2. Architecture

The idea behind SCFP is to encrypt programs at compile time using a sponge-based AE cipher. Decryption is then performed within the CPU, instruction by instruction, just

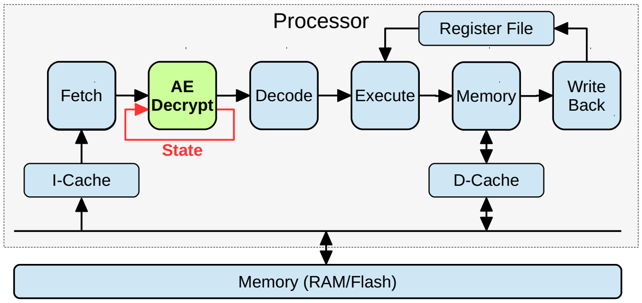

Figure 1. High-level system architecture of a classic RISC processor which has been extended for SCFP with a sponge-based AE decryption stage.

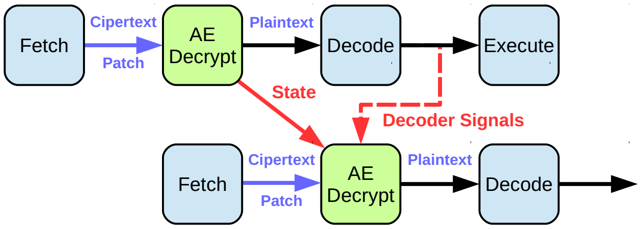

Figure 2. Data dependencies between two consecutive instructions within a processor pipeline when SCFP is implemented. The decoder signals can optionally be fed back.

in time for execution. At its heart, the sponge-based AE cipher uses an internal state z, which provides the foundation for the CFI protection in SCFP. This state accumulates information about all the processed instruction ciphertexts, which enforces that correct decryption is only possible iff all previous instructions have also been genuine. Conceptually, with every processed instruction ciphertext C, the plaintext instruction P as well as a new internal state z' are derived from the current state z using a permutation f following $(P|z') = f(C|z)$ ($|$ denotes concatenation). As a result, the correctness of the plaintext instructions that get executed by the CPU does not only depend on the fetched input (i.e., ciphertext), but also on the history which has been accumulated within the internal state of the cryptographic primitive.

If either the state (e.g., through a CFI violation or clock glitch) or the ciphertext (e.g., through manipulation in memory) is erroneous, correct decryption is not possible anymore and pseudo-random instructions are produced as plaintext. We consider the respective execution of random instructions a secure processor state for two reasons. First, the probability of random code which is generated by SCFP to be meaningful is extremely low, especially when attack gadgets of multiple instructions are required. Second, attackers neither have control over the random instructions being executed, nor can attackers observe what the plaintext instructions are during random execution. This effectively hampers any attacker attempts to execute harmful code. Besides, we will later show that SCFP supports the detection of random code execution to add error handling as desired.

From a processor architectural point of view, the ideal location within the processor pipeline for performing the decryption is between instruction fetching and decoding, as shown in Figure 1. The instructions are transferred from the fetch to the decode stage exactly in the execution sequence, which also matches the desired decryption sequence of SCFP. As the decode stage is the first to need plaintext instructions, performing decryption right in front of the decoder is in fact also the latest possible point for inserting SCFP and effectively minimizes the number of components with plaintext code access to the decode stage itself. All the other components, like peripherals, main memory, various caches, memory buses, and even the fetch unit, operate on encrypted code only.

Figure 2 depicts the instruction-data dependencies between the different pipeline stages for the processor from Figure 1. A traditional scalar processor with a pipelined architecture only has dependencies between the different stages (visualised horizontally) but not across multiple instructions (visualised vertically). The processor basically decodes each instruction completely isolated from other instructions. Dependencies between instructions are solely a result of data dependencies in the program (e.g., via the register file) which can lead to pipeline hazards and stalls. Extending the pipeline with an AE decryption unit breaks this isolation between instructions and introduces an additional dependency via the cipher state.

For scalar processors, it is additionally possible to feed the data independent decoder signals of each executed instruction back into the cipher. Such feedback extends authenticity protection up to the pipeline's execute stage and can, for example, be used as a link to fault countermeasures in the ALU. Note however, that the SCFP approach is not limited to scalar processors. Superscalar microarchitectures can also be protected using SCFP with a coarser granularity, e.g., decrypting multiple instructions instead of individual instructions in one block.

2.3. Authenticated Encryption and Control Flow

Sponge-based authenticated encryption schemes use a single internal cipher state for both encryption and authentication. This common state leads to the nice property that the mapping between each encrypted and plain instruction depends on the actual values of all previously processed instructions. Hence, to be able to encrypt a program such that it can be executed on a processor that implements SCFP, the exact sequence of executed instructions needs to be known at compile time. However, exactly this property makes the combination of authenticated encryption with control flow challenging.

More concretely, at compile time, the exact instruction sequence can only be determined for a very limited number of programs. Basically, only programs that have a completely data independent control flow (e.g., no data dependent branches) can be trivially supported. Additionally, even genuine and intended code reuse (e.g., loop bodies or functions) is not easily possible anymore. This is due to the

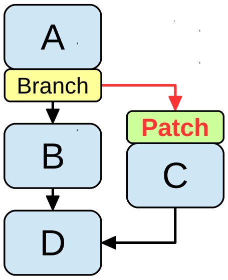

Figure 3. Simple example of patching the CFG of an if-then-else construct in SCFP.

fact that after encryption, the ciphertext is fixed and correct decryption of an instruction is only possible given the correct unique cipher state (and thus execution history). Placing the sponge-based authenticated encryption scheme into the processor pipeline therefore provides a solid foundation for SCFP and thwarts code reuse by default.

The main idea to allow specific code reuse in SCFP and to make SCFP applicable to general programs is to deliberately introduce collisions into the internal state of the cryptographic primitive. These state collisions are conceptually a white listing of permitted control flow transfers and have to be introduced exactly at the required positions in a program. Note that these deliberately introduced collisions do not weaken the security of the cryptographic primitive.

The simplest and most efficient way to generate the required state collisions is to inject additional metadata as correction terms into the cipher state at certain points during the execution of the program. We denote this process of deliberately adjusting the AE state as *patching* and the involved constants as *patch values*. Via patching, we effectively cancel out divergences in the cipher state which originate from taking different valid paths through the Control-Flow Graph (CFG). As the result, correct decryption of a program under SCFP is only possible as long as the execution adheres to the statically determined CFG.

It has to be noted that patching must be implemented as a differential update of the AE state. Otherwise, if patching was implemented by simple replacement, patching would destroy all the history which had been accumulated into the state. Besides, the patching process must be able to modify the full sponge state in order to create arbitrary collisions.

2.4. Patch Handling, Placement and Calculation

The patch values in SCFP are conceptually very similar to the justifying signatures in the soft error and fault attack countermeasures based on Continuous Signature Monitoring (CSM) [43], [44]. Therefore, also similar implementation techniques can be used to find suitable patch locations as well as to determine the concrete values of the patch constants.

More concretely, the task of the patch values in SCFP is to introduce cipher state collisions at the merge points in the CFG of the program. Hence, all differences which originate from traversing the statically determined CFG along runtime data dependent paths have to be compensated. An example for patching a simple if-then-else construct is shown in Figure 3. There, a patch value is injected into the cipher state before the execution of Basic Block (BB) C (i.e., on the red CFG edge) such that the state at the beginning of BB D is the same, regardless of whether the blocks A and B, or the blocks A and C (incl. the patch) have been executed.

The exact way how such a patch value is encoded into the program and how patches are processed during runtime strongly depends on the concrete implementation of SCFP and is highly Instruction-Set Architecture (ISA) specific. However, an intuitive way to implement and think about cipher state patching is to consider the patch values as part of specialized control-flow instructions. Similar to immediate operands in standard instructions, the patch values are part of the instruction encoding and get fetched like regular code by the processor during execution.

From the toolchain perspective, implementing SCFP consists of two steps. In the first step, during compilation, patches have to be inserted at the correct positions into the program by emitting suitable instructions with patch support. In the second step, at link time or in a post processing phase, the program binary has to be encrypted and the correct patch values have to be inserted into the binary (i.e., similar to relocations).

For a program which comprises only branches and direct calls, a functional solution for patch placement during compilation can be obtained by looking at the undirected CFG of the full program. Every cycle in this graph has to be broken by introducing a patch for the cipher state. Therefore, the minimum number of patches and possible positions can be obtained by comparing the CFG with its spanning tree. Taking the function call graph into account, this approach is also applicable to indirect and recursive function calls. Unfortunately, comparably expensive whole program analysis has to be performed to acquire the mentioned graphs.

Nevertheless, also compilation in multiple translation units can be supported with SCFP when a well-defined patching convention is established around function calls. Similar to a regular calling convention, having a patching convention allows to correctly place patches in every function of the program in isolation. Within each function, it is then typically sufficient to always patch when a branch is taken as shown in Figure 3. Additionally, to cope with recursion, it has to be ensured that at least one patch is performed before the recursion is entered. Note however that the simplicity of the patching convention, compared to the graph based approach, comes at the cost of an increased number of patch values.

To illustrate the concept, in the following, an exemplary patching convention for direct and indirect function calls is presented.

2.4.1. Direct Calls. Every function which gets directly called from more than one call site within a program necessarily requires patching. In particular, at least $n-1$ patches are required when n call sites exist. Interestingly, this situation is also similar to the direct branch example

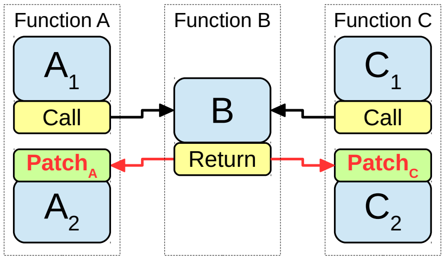

Figure 4. Example of a simple patching convention for direct function calls. Function B can be called from both, function A and C.

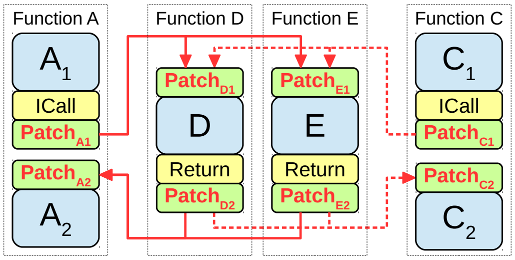

Figure 5. Example of a simple patching convention for indirect function calls. Functions A and C can call both, functions D and E at runtime.

in Figure 3 where one patch is required since two paths in the CFG lead to BB D. However, placing patches at every call site except one again requires access to the full program during compilation. To relax this constraint, at the cost of one additional patch per function, patching can simply be performed on every call site as shown in Figure 4. In this example, $Patch_A$ has to be applied when the control flow returns from function B to function A. Returning from function B to C uses $Patch_C$, respectively.

Note that, in most cases, having one patch per direct call is sufficient regarding both functionality and security, because typical ISAs perform direct calls relative to the program counter. In this case, the program counter relative offset is part of the function call encoding and is different for each call site. This implies a different, internal SCFP state for each call site. As a result, besides the required state collisions at the call and return edges of the direct function call, there are no other, undesired collisions being introduced to the program.

In general, it does not matter whether the patch value is applied at the return operation or the call operation, as long as it is done consistently and aligned with the way branches are patched. Applying patches is therefore possible either after branches and on returns (as shown in Figure 3 and Figure 4), or before merge points of branches and during calls. In fact, when looking at the CFG of the whole program, function returns are simply branches and function calls are merging points.

2.4.2. Indirect Calls. Similar to direct function calls, also indirect function calls require patching. However, determining the exact function which gets called at runtime by an indirect function call is not always possible at compile time. Moreover, often also multiple different functions get called from the same indirect call site during the runtime of a program (e.g., comparison callback of `qsort`). Therefore, the best one can do with static CFI such as SCFP is to determine a, possibly over approximated, set of potential call targets and to enforce that only calls to functions in this set are possible at runtime.

Our current approach to implement indirect function calls and returns with SCFP is shown in Figure 5. In total, two patch values have to be applied on every indirect control-flow transfer. The idea of this scheme is to use the first patch (e.g., $Patch_{A1}$) to reach a constant cryptographic intermediate state, which is then updated to the actual entry state of the called function using the second patch value (e.g., $Patch_{D1}$). The constant intermediate state can be freely chosen at encryption time and permits to restrict indirect calls to targets which were encrypted for the same intermediate state.

In summary, for the patching convention in Figure 5, two patch values are required for every indirect function call site as well as for every function which can be called indirectly. At runtime, in total four patches get applied for every indirect function call.

At the first glance, using four patch values for one indirect function call may seem excessive given that two patches would already suffice to build a functioning CFI scheme. However, using less patch values necessarily introduces undesired collisions into the SCFP state which weakens the confidentiality and authenticity properties of the scheme.

2.5. Initial State Derivation

In sponge-based AE ciphers with known permutation, the initial state is comparable to the key in regular encryption schemes. It is common to derive this initial state z_I from a secret key k and public nonce N by applying their permutation f (e.g., $z_I = f(N|k)$). Conceptually, we recommend using a similar approach for deriving the initial state in SCFP. This ensures that, even when k is a device-specific fixed master key, every program for that device is still encrypted under a different initial state. Optionally, additional information like, for example, the start address or the program vendor can be used during the derivation of the initial state.

Note that binding initial states to the machine key also serves as software diversification. Namely, in case successful exploits against SCFP should be found in a program on a certain device, they cannot simply be transferred to other devices executing the same program.

2.6. Interrupt Handling

Unlike regular function calls, which are performed at precisely defined points during program execution, interrupts can occur at virtually any time. It is therefore impossible to determine a unique differential update value for all the states which permit to call an interrupt handler. We cope with this problem in SCFP by treating interrupt handlers similar to the initial program entry point. Therefore, we derive a new AE cipher state to re-initialize SCFP when entering the interrupt handler. On the other hand, the SCFP state that is active before entering the interrupt handler is, similar to the old program counter, saved in an internal processor register. For the operating system, the SCFP state is therefore simply one additional register which has to be saved and restored during context switches. Note however that, to ensure that the confidentiality of the SCFP state is maintained at all times, the old state value which is stored in the processor register should be encrypted or similarly protected.

Implementing interrupt handling in this way effectively separates the protection of interrupt handlers from the regular code. This means that interrupts can be processed successfully even when a regular program executes pseudo random instructions due to an attack. On handler entry, this separation is desirable as it allows us to recover from errors in software as well as to perform scheduling of programs via the operating system. On handler exit, on the other hand, we want to propagate errors occurring during the execution of the interrupt handler into the execution of the regular code.

We achieve this behavior by enforcing that the internal SCFP state has a predefined secret value when returning from the interrupt handler. Similar to the state derivation on interrupt entry, the secret handler exit state can, for example, again be computed from the key, the nonce, and the address of the interrupt handler. When returning to the regular code execution, the hardware can then simply combine the current state z, the expected exit state e, and the state from before the interrupt entry z_{entry} from the register to calculate the next state z', e.g., $z' = z \oplus e \oplus z_{entry}$. By doing so, the entry value is only restored ($z' = z_{entry}$) correctly as next state if also the handler execution has been genuine ($z == e$).

2.7. Fast Error Recovery

As SCFP ensures security even without explicit fault checks, SCFP eliminates the existence of a single point of failure. Namely, the probability of random code execution in SCFP to be meaningful is extremely low. While this is one major benefit of SCFP, it may still be desirable to provide a timely way to perform error recovery after the processor started to execute a random instruction sequence. Interestingly, the execution of pseudo random instructions in the error case already provides one way to permit error recovery since the processor is able to identify invalid instructions. The concrete detection probability follows a geometric distribution and can be computed when the ISA of the processor is known. More concretely, given the probability p_{inv} for a random instruction to be invalid, the expected detection latency l is computed as $l = 1/p_{inv}$. However, considering that modern ISAs are often quite dense, recovery latency can be comparably high.

Faster recovery can be achieved when additional redundancy bits are verified on the execution of every single instruction. Sponges permit to implement this additional integrity verification in an efficient and secure way by simply checking the desired amount of state bits. No additional permutation calls, but only a marginally bigger permutation is required. The strength, i.e., the number of bits, for this verification can be freely chosen, but is typically rather weak for a single instruction. However, the continuous nature of this check compensates for this weakness quite fast. In general, the number of asserted bits allows to trade off between the code size overhead and the recovery latency.

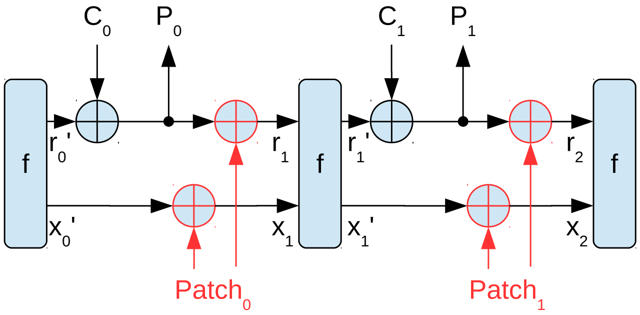

Figure 6. Decryption using a duplex construction similar to the one used in SpongeWrap.

3. Sponge Constructions for SCFP

SCFP relies on a scalable and strong sponge-based authenticated encryption cipher. This section introduces two eligible sponge-based constructions and presents arguments for their security as well as guidelines for parameter selection.

3.1. Constructions

Cryptographic sponges have become quite popular since Keccak has been announced as the winner of the SHA-3 competition. However, sponges can also be used to build other cryptographic primitives. The Keccak designers themselves, for example, already proposed an AE mode called SpongeWrap [11] early on and still pursue the idea with Keyak [14] and Ketje [13] in the ongoing CAESAR competition [10]. The other numerous sponge-based submissions [3], [8], [23], [28], [36], [37] to the competition further support this research direction.

Considering the success and general properties of sponges, the following discusses two sponge-based constructions which have been adapted to support the patching of SCFP. This approach allows us to profit from the substantial amount of cryptanalysis performed on the various sponge constructions and the underlying permutations. In general, we therefore recommend well-analyzed permutations like Keccak-p. However, a more detailed discussion on suitable instantiations of SCFP, including permutations, can be found in Section 4.

3.1.1. SpongeWrap-like Decryption Mode. The first construction, shown in Figure 6, is based on the duplex construction, which has been introduced and proven to be secure in [11]. This duplex construction is used in SpongeWrap for both encryption and decryption. When executing strictly

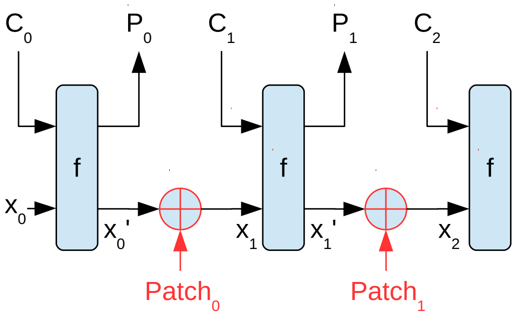

Figure 7. Decryption in an APE-like construction.

sequential code, where no patching is required ($Patch_i = 0$), AE on the instruction stream is identical to SCFP. However, for generic code SCFP must also implement branching. Therefore, additional support for the injection of patch values has to be added to the construction. Both the rate and the capacity of sponge make up the previously described SCFP state z and must be modifiable by such a patch.

From the security point of view, these patch values can be considered as Associated Data (AD). AD denotes data that is authenticated, but not encrypted. It has been shown by Mennink et al. [35] as well as Sasaki and Yasuda [38] that it is secure to absorb AD into the capacity of a keyed sponge. Considering that the construction in Figure 6 is a keyed full-state duplex sponge construction, it is therefore secure to inject the patch values into the capacity. Updating the rate with the patch is secure as well given that the rate is under the control of an attacker via the ciphertext in any case.

The SpongeWrap-like construction has two neat features. First, its implementation is comparably simple since encryption is identical to decryption. Second, the construction provides great flexibility as it permits to calculate and place patch values on arbitrary places in the CFG. However, there are also some drawbacks which have to be considered. For example, an attacker might be able to precisely control the first fault given that errors in the ciphertext directly propagate into the plaintext ($\Delta C_i = \Delta P_i$). In a known plaintext attack, this might even permit to inject one specific instruction before the plaintexts of subsequent instructions are randomized.

Note also that, if the control-flow merges at instruction i and patches are not applied directly before the merge point in the control-flow graph (i.e., $Patch_{i-1} = 0$), all instructions directly preceding the merge point (P_{i-1}) have to be identical. This is due to the fact that, as soon as the instruction at the merge point is fixed (i.e., P_i and C_i), also the plaintext of the predecessor P_{i-1} is determined by the dependency over the rate part of the sponge ($P_i = C_i \oplus f_r(P_{i-1}|x'_{i-1})$). However, this link can be broken by performing patches solely at merge points of the CFG instead of placing them freely.

3.1.2. APE-like Decryption Mode. The second construction is inspired by another AE mode of operation called APE [4]. The layout of the APE construction itself is similar to the duplex construction in SpongeWrap. However, APE is not inverse free, i.e., the inverse permutation f^{-1} is needed for encryption when the permutation f is used for decryption. Moreover, the indices of the cipher- and plaintexts have been rearranged compared to SpongeWrap. Namely, in APE, the plaintext P_i, corresponding to a ciphertext C_i, is calculated as $P_i = C_{i+1} \oplus f_r(C_i|x_i)$.

The APE-inspired mode we propose in Figure 7 is calculated as $P_i = f_r(C_i|x_i)$ and modifies APE in two ways. First, P_i's dependency on C_{i+1} is removed. This solves the problem of the SpongeWrap-like mode where attackers can inject one specific instruction if they know the original value. Moreover, this modification makes our construction behave more like a block cipher than a stream cipher. Second, patching capabilities for the capacity are introduced to make the construction suitable for SCFP. Note that the APE-like construction is superior to the SpongeWrap-like mode in this regard. It only needs patching of the sponge's capacity which corresponds to the SCFP state z. On the other hand, the sponge rate is not chained any more.

The main drawback of the APE-like construction is that it is less flexible, because the position of patches is fixed. Patches always have to be positioned at branching points in the control-flow graph. This is a result of the encryption that has to be performed in inverse direction to the decryption (i.e., inverse to the execution sequence).

3.2. Parameter selection

It has been shown that the duplex construction [11] as well as the APE construction [4] are secure against generic attacks which do not exploit properties of the underlying permutation. The complexity of such attacks is lower bounded by $2^{x/2}$ and depends on the capacity size x. To provide s-bit security, x must thus be chosen as $x \geq 2 \cdot s$.

The size of the sponge rate depends on the actual implementation. The majority of the rate is needed for the decryption of the instructions. The instruction size i depends on the ISA and is typically 16 or 32 bits. However, additional rate bits may be used for fast error recovery. To enable fast error recovery of n bits without leaking parts of the plaintext nor reduction of the security, a rate of $r = i + n$ bits, and hence a permutation size of $b = i + n + x$ bits is needed.

The proofs in [5], [12] show that also smaller capacity sizes can result in cryptographic security. These results can be used to either reduce the permutation size while maintaining the security level, or to increase the security of a fixed permutation. However, a limit on the data complexity, which strongly depends on the actual implementation, is required to benefit from these refined proofs. We therefore refrain from proposing parameters based on the proofs in [5], [12] and leave the exploitation to implementers knowing the respective system characteristics.

4. Instantiations

The flexibility of SCFP allows to tailor its protection level to the needs of the respective application by choosing a suitable permutation for the sponge-based AE scheme. In this section, we hence introduce three different instantiations of SCFP. First, AEE uses a large, unkeyed permutation to yield confidentiality and authenticity of the program binary as well as CFI to prevent fault, code reuse, and code-injection attacks. Second, IE uses a small, unkeyed permutation to form a lightweight CFI scheme to prevent code-reuse and fault attacks only. Third, the use of a small, keyed permutation in AEE-Light yields small overhead, CFI and IP protection in trade for weaker authenticity. We first discuss the properties of unkeyed permutations used in AEE and IE, and then proceed with keyed permutations utilized in AEE-Light.

4.1. Unkeyed Permutations

When instantiating SCFP with unkeyed permutations, the cryptographic security properties of SCFP are solely determined by the size of the sponge capacity x. Neglecting the proofs in [5], [12], a security level of s bits requires a sponge capacity of $2s$ bits. However, these are generic results without consideration of the actual application.

In particular, the cryptographic security level s is mainly determined by collisions in the cryptographic state. These can generically be exploited in Time-Memory Trade-Off (TMTO) attacks with birthday bound complexity $2^{x/2}$ and eventually allow state recovery and thus IP theft or forgery. However, to perform these TMTO attacks, the attacker must also be able to observe the output of the sponge, which is not the case for SCFP. Namely, the instructions decrypted by SCFP are internally processed by the processor and never directly revealed to the attacker. As a result, the complexity for state recovery for SCFP is 2^x in practice. In a similar way, the probability of arbitrary state collisions in a binary encrypted and authenticated with SCFP is in general determined by the birthday bound, i.e., $2^{-x/2}$. However, attackers do not have access to the decrypted instructions and the internal state when using SCFP. Attackers are thus unable to observe and detect internal state collisions. Hence, meaningful exploitation of internal state collisions for SCFP is equivalent to state recovery and has complexity 2^x as well.

Software Attack Complexity. These considerations have a significant impact on the actual attack complexities for code injection and code-reuse attacks when SCFP is in place, i.e., the CFI properties of SCFP. Namely, attackers performing code injection or code-reuse attacks require precise control over the executed instructions to succeed. For example, attackers can modify a single instruction with success probability 2^r, but will neither be able to observe whether they hit the right instruction, nor be able to modify the internal state such that all successive instructions remain the same. This means that the attacker must adapt all successive instructions too, because the processor will otherwise execute random instructions. However, precise manipulation of n instructions has even higher complexity, namely $2^{n \cdot r}$. Alternatively, attackers can try to learn the internal state to correctly encrypt and inject their own program. However, this has complexity 2^x. A different example are modified jump targets in code-reuse attacks. As attackers manipulate addresses to jump to well-defined instructions in the binary, the x-bit patch values must be adapted accordingly. However, finding a correct patch value has complexity 2^x too.

Fault Attack Complexity. The CFI properties of SCFP also increase the attack complexities for fault injection attacks that manipulate control flow or instructions prior to the decode stage. For example, simple instruction skips or repetition have a success probability of 2^{-x}. The same probability applies to arbitrary control-flow errors, e.g., caused by a randomly faulted program counter. On the other hand, performing a specific control-flow transfer via faults is as hard as forcing the x capacity bits and the program counter (e.g., 32 bits) to the desired value. However, this is non-trivial since the sponge state is secret and must be extracted or brute forced first. Furthermore, being able to control that many bits precisely is quite hard in practice.

Instead of altering the control flow, fault attackers can also try to manipulate code by injecting bit flips. For example, attackers can use clock or power glitches to inject random bit flips into code. However, it takes roughly 2^r tries to hit one specific instruction with random bit flips. Therefore, another approach is to use a small and limited number of precise bit flips in the fault attack instead. Yet, exploiting precise bit flips in the encoded value is as hard as utilizing a differential characteristic of the permutation. Only precise bit flips in the plain instruction can be exploited directly. However, regardless of whether random or precise bit flips are injected, bit flips in code modify the sponge state as well. This randomizes the sponge output of all subsequent instructions and therefore prevents further exploitation. Moreover, SCFP can also protect the plain instructions against fault attacks as well by feeding the decoder signals back into the sponge state.

Depending on the concrete security level s and choice of sponge parameters r and x, we identify two different types of SCFP instances using unkeyed permutations. First, AEE denotes instances with cryptographic security levels, i.e., at least 80 bits, that offer CFI as well as confidentiality and authenticity of software IP. Second, IE denotes instances below cryptographic security levels to enforce CFI only.

4.1.1. Authentic Encrypted Execution.

AEE features cryptographic security levels for encrypting and authenticating code. This automatically defeats adversaries which wiretap the communication to the external memory chips without any need for further code encryption and/or authentication. Moreover, software attacks are made harder too. As other CFI schemes, AEE hampers return- and jump-oriented programming attacks. The strong encryption and authentication further mitigates both code injection and code-disclosure attacks. AEE is therefore a replacement for established software attack countermeasures like DEP/W^X, CFI, and R^X. In addition, by enforcing CFI AEE also prevents fault attacks on the processor chip that aim at instruction or control-flow manipulation.

From a cryptographic perspective, AEE requires a permutation size of at least 192 bits to yield 80-bit security for a 32-bit instruction set. One suitable permutation to instantiate AEE hence is Keccak-p[200,12] [14] with 200-bit state size and 12 rounds as used in Keyak. The exceeding 8 bits increase the capacity and thus the security level to 84 bits. However, as elaborated before, the specifics of AEE result in a complexity of 2^{168} for state recovery, control-flow hijacking, and fault attacks on control flow. Similarly, a single instruction can be successfully manipulated from software or using fault attacks with complexity 2^{32}, but the internal 168-bit state will cause the execution of random instructions afterwards.

4.1.2. Infective Execution. Contrary to AEE, IE uses a small permutation and thus, from a cryptographic point of view, cannot provide a strong level of security. In particular, IE behaves like a context-sensitive Instruction-Set Randomization (ISR) rather than authenticated encryption. IE thus fails to ensure confidentiality and authenticity of software IP. However, the parameterization of IE forms a practical CFI scheme that considerably complicates code-reuse attacks as well as fault attacks on the processor chip itself. Yet, the concrete instantiation of IE is highly application specific.

For a 32-bit ISA, IE can, for example, be instantiated with 50-bit state size and the Keccak-p[50,12] [14] permutation (i.e., 12 rounds as in Keyak). Using two bits for fast error recovery gives a sponge rate $r = 34$ bits and a sponge capacity $x = 16$ bits, which also corresponds to the size of the patch values. From a cryptographic perspective, this IE instance yields merely 8-bit security. However, the probability for successful code injection and manipulation of control flow still is 2^{-16}.

The main drawback of IE is that an attacker with access to the encrypted binary can easily perform state recovery offline, in our example with complexity 2^{16}. State recovery eventually breaks the CFI property of IE for software attackers. Namely, a software attacker knowing the secret, internal IE state can compute correct ciphertexts and patch values, and inject these into the code from within software when performing code injection or reuse attacks. However, the complexity of physical fault injection on the processor chip itself is still high enough for the parameterization of IE. Nevertheless, to ensure CFI for software attackers as well, access to the encrypted binary must be limited. While this restricts the attacker compared to the original threat model in Section 2, access control can easily be enforced using two different mechanisms: (1) by using execute-only memory, software attackers lose online access to the encrypted binary, and (2) by storing the binary in on-chip memory, attackers with physical access cannot read the encrypted binary any more. As a result, IE is particularly interesting for tiny IoT devices without external memory and for smart cards. Note however that state recovery, code analysis, and wide-spread deployment of attacks can easily be mitigated by using a different seed for IE on every device as this causes the internal states, patch values, ciphertexts, and positions of state collisions to change. Moreover, note that the probabilities for manipulating control flow stated above are enough to enforce CFI and are indeed in the range of entropy estimations of other techniques to prevent code-reuse attacks, e.g., software diversification [18].

4.2. Keyed Permutations

AEE enforces its security properties by using a sufficiently large permutation and thus capacity. However, a sponge capacity providing cryptographic security levels also implies larger AEE patch values and thus memory overhead. On the other hand, IE yields lower memory overhead by using a small permutation, but cannot sufficiently protect software IP and its authenticity. For this reason, an SCFP instance with low memory overhead, but with similar security properties as AEE, is desirable. One approach to tackle this problem are keyed permutations.

When using a keyed permutation, the security of SCFP does not only depend on the sponge capacity x, but also on the security level s_p of the permutation itself. As for AEE and IE, the authenticity when using keyed permutations is determined by the sponge capacity x, i.e., the authenticity level is $x/2$ bits. However, the complexity of learning the plaintext of the encrypted binary is 2^{x+s_p} and thus also depends on the security guarantees of the permutation with respect to the permutation key.

4.2.1. Authentic Encrypted Execution Light. We build on this observation and introduce AEE-Light to denote SCFP instances based on keyed permutations. AEE-Light offers the same security bounds as AEE with respect to authenticity and CFI. For example, control flow hijacking and fault attacks on control flow have complexity 2^x, whereas successful injection of a single instruction has complexity 2^r. On the other hand, successful recovery of the software IP or the internal state from the encrypted image has complexity 2^{x+s_p}. By using a permutation with sufficiently high security level s_p, the confidentiality of software IP is hence guaranteed and state recovery, code injection, and meaningful forgery are prevented. In particular, even if an attacker recovers the x-bit internal state, meaningful injection or forgery of more than one instruction still has complexity 2^{s_p} as the permutation key is unknown to the attacker.

For 32-bit instructions, a suitable choice for the keyed permutation is the 64-bit block cipher PRINCE [16], which uses a 128-bit key to offer $s_p = 96$-bit security. This results in a sponge capacity $x = 32$ bits. State recovery using this AEE-Light instance has complexity 2^{128} and is thus infeasible. This effectively protects the software IP and prevents both code injection and analysis. Contrary to that, IE from before uses a similarly small permutation but cannot guarantee any of these features without further techniques to hide the encrypted binary. However, while the cryptographic level of authenticity guaranteed by this instance of AEE-Light is only 16 bits, meaningful code-reuse attacks and forgery are much harder. Namely, the expected complexity to find the correct patch value is 2^{32}, which is enough to enforce CFI and to prevent code-reuse and physical fault attacks. Besides, the $s_p = 96$-bit security of the permutation further hardens any attempts to tamper with the software binary in a meaningful

TABLE 1. Examples of SCFP instances for a 32-bit ISA and the respective attack complexities.

	Parameters [bit]		Cryptographic	Attack Complexity [bit]				
Permutation	x	s_p	Security [bit] [a]	CIA [b]	CRA [c]	ESIP [b]	FAIS [c]	Type
Keccak-p[200,12]	168	—	84	168	168	168	168	AEE
Keccak-p[50,12]	16	—	8	16	16	16	16	IE
PRINCE	32	96	16	128	32	128	32	AEE-Light

[a] Random collisions in the capacity x have birthday-bound complexity and limit the achievable authenticity, i.e., $x/2$ bits.
[b] Requires the recovery of capacity and permutation key, i.e., $x + s_p$ bits.
[c] Requires to find and inject the correct patch values, i.e., x bits.
[d] CIA: Code-Injection Attacks; CRA: Code-Reuse Attacks; ESIP: Extraction of Software IP; FAIS: Fault Attack with Instruction Skips

way. In particular, even though one single instruction can be manipulated with complexity 2^{32}, meaningful modification of multiple instructions is significantly harder since both the internal AEE-Light state and the permutation key are unknown to the attacker.

4.3. Discussion

Table 1 summarizes the exemplary instances of AEE, AEE-Light, and IE. In detail, Table 1 shows the respective cryptographic security as well as the actual attack complexities for code injection, code-reuse attacks, extraction of software IP, and instruction skips using fault attacks. AEE is the strongest variant with 168-bit security for all considered attacks. At the further end, IE is the smallest variant and offers merely 16-bit security for the mentioned attacks. However, this suffices to enforce CFI and prevent code reuse as well as fault attacks on control flow when the code binary remains hidden. As a trade-off between these two, AEE-Light uses keyed permutations to simultaneously attain small 32-bit patch values, i.e., low memory overhead, and good security properties. In particular, AEE-Light provides 128-bit security in terms of IP recovery and code injection, whereas its security level with respect to code reuse and control-flow fault attacks is 32 bits and thus sufficiently high for CFI.

5. Evaluation

While SCFP protects software and its execution from a large set of attacks, SCFP also has an impact on performance in practice. In this section, we determine this performance impact by analyzing binary size increase and execution time overhead of AEE-Light implemented on a RISC-V processor. Our results demonstrate the practicality of SCFP with a size overhead of 19.8 % and a performance overhead of 9.1 % on average.

5.1. Architecture

The basis of our evaluation hardware is the RI5CY core which is part of the PULPino [24] System on Chip (SoC). RI5CY is an in-order implementation of the RISC-V ISA [42] with four pipeline stages. More concretely, our processor supports the RV32IM subset of the ISA with privilege architecture version 1.9 and has been extended with optional SCFP support. When SCFP is enabled, AE decryption is performed in an additional pipeline stage between instruction fetching and decoding. The processor therefore has four pipeline stages when SCFP is disabled and five when SCFP is active.

As SCFP instance, an AEE-Light configuration with a 64-bit sponge state and the PRINCE block cipher as a keyed permutation in an APE-like sponge construction has been implemented. Since our processor supports both normal code execution and SCFP-protected execution, seven custom control-flow instructions have been added. These new instructions have similar semantics as the existing conditional branch (`BEQ, BNE, BLT, BLTU, BGE, BGEU`) and jump (`JAL, JALR`) instructions, but additionally apply patch values as needed. In particular, our protected conditional branch instructions (`BPEQ, BPNE, BPLT, BPLTU, BPGE, BPGEU`) always apply a patch when the branch is taken. On the other hand, our protected jump instructions (`JALP, JALRP`) apply either one or two patch values depending on the alignment of the target address. The respective patch values used by these instructions are embedded within the encrypted binary as constants in the `.text` section and are located next to the branch/jump instruction they belong to.

We furthermore designed a SoC, containing our modified processor with a design frequency of 100 MHz, in a UMC 65 nm technology, which is currently manufactured as an Application Specific Integrated Circuit (ASIC). Within this ASIC, our processor occupies 90 kGE of which around 32 % are due to the SCFP implementation. With 91 %, the majority of this overhead is due to the fully unrolled, single cycle PRINCE implementation. Note that extending the RISC-V core with SCFP did not change the target frequency of 100 MHz.

5.2. Software Translation

Our toolchain, which generates binaries for our custom processor in AEE-Light mode, is at the moment rather simple. We employ the standard RISC-V GNU toolchain which we only extended with assembling support for our custom instructions. Additionally, a post-processing tool has been developed which consumes the final elf binary in order to perform the encryption of the code and to fill in the required patch values.

Since the C compiler has not been extended with SCFP support, this toolchain natively can only handle assembler code which contains placeholders for the patch values and uses our protected control-flow instructions. However, due to the way we designed our instruction-set extension, we can quite easily support the protection of C programs via simple textual replacement of instructions on the assembly level.

More concretely, at the moment, when compiling C code for our processor, we first compile the C code to assembly, where we replace all ordinary control-flow instructions through the protected counterparts and embed `NOP` instructions as placeholders for the patch values. The resulting assembly files are then assembled and linked. Finally, the resulting elf file is processed using our post-processing tool which emits the encrypted binary.

This simple flow already suffices to demonstrate the practicality of SCFP. Note however, given that the compiler is not aware of SCFP, also the code has not been optimized for the correct cost model (e.g., loops get more expensive due to the patching). We therefore consider the proper integration of SCFP into the compiler as future work and expect that even better performance can be achieved with an optimized compiler which is aware of SCFP.

5.3. Results

We used our software toolchain to instrument, compile and encrypt a set of C benchmarks to evaluate our implementation of AEE-Light. As benchmarks, we used several programs from the PULPino repository [24]: AES in CBC mode (`aes_cbc`), a 2-dimensional matrix convolution (`conv2d`), 100 runs of `dhrystone`, a finite response filter (`fir`), a fast Fourier transform (`fft`), and an implementation of the inflection point method (`ipm`). Moreover, we used two implementation variants of the elliptic curve point multiplication (SECP192R1) that were internally available at our department. Both `ecc` and `ecc_opt` are pure C implementations targeted at microcontrollers. However, while `ecc` uses a generic implementation of the underlying multi-precision integer arithmetic, the multi-precision integer arithmetic in `ecc_opt` uses completely unrolled loops and only works for the specific elliptic curve. We compiled all programs at optimization level `-O3`. Since the manufactured ASIC is not yet available at our department, all runtime values have been determined using cycle accurate HDL simulation.

Table 2 shows the results of our evaluation of code size and execution time. In particular, Table 2 compares the unprotected, standard executables of our benchmark programs with the executables protected and encrypted via our instance of AEE-Light. Both program versions have been executed on our modified processor which features either a four stage (i.e., baseline) or a five stage (i.e., AEE-Light) pipeline.

TABLE 2. EVALUATION RESULTS OF AEE-LIGHT IN HDL SIMULATION.

	Code Size (`text + data`)		Runtime	
	Baseline	Overhead	Baseline	Overhead
	[kB]	[%]	[kCycles]	[%]
`aes_cbc`	10.0	14.8	43.4	9.5
`conv2d`	4.6	25.6	5.4	4.8
`dhrystone`	7.5	20.1	50.6	14.4
`ecc`	9.3	21.0	4282	9.2
`ecc_opt`	9.6	20.1	3032	3.8
`fir`	5.5	20.9	24.0	9.5
`fft`	7.1	16.8	45.6	7.0
`ipm`	8.8	19.4	4.5	14.9
Average		19.8		9.1

For our set of benchmarks, it shows that the overhead in code size due to the inserted patch values ranges between 14.8 % and 25.6 % and averages to 19.8 %. This overhead is mainly affected by the number of branches and function calls in the binary. On the other hand, the runtime overhead ranges between 3.8 % and 14.9 % and averages to 9.1 %. This runtime overhead is significantly lower than the code size overhead and mainly depends on the number of branches and function calls that are effectively taken during runtime. This becomes especially visible for the two implementations of elliptic curves. Namely, as the inner loops are unrolled in `ecc_opt`, the number of executed branches drops massively from 170 k taken branches to 20 k and hence the runtime overhead for `ecc_opt` is much lower than for `ecc`. On the other hand, `ecc` and `ecc_opt` yet have very similar code size and code overhead.

Summarizing, the figures in Table 2 indicate that AEE-Light can protect against a wide range of IoT threats with practical overheads. In addition, our AEE-Light instance features a low expected detection latency for execution errors of merely two cycles, because the RISC-V instruction set we implemented is quite sparse ($\sim$75 % invalid encodings). However, note that using a more sophisticated toolchain for generating AEE-Light executables can lead to even smaller overheads.

6. Related Work

In general, other CFI schemes as well as work on Trusted Execution Environments (TEEs) can be considered related to SCFP. However, most techniques provide vastly different security guarantees given that they are designed to counteract software attacks only.

Control-Flow Integrity. Numerous CFI schemes have been introduced in the last 30 years. However, techniques providing fine-grained CFI and code integrity, as required to detect physical attacks, are quite rare [22].

SOFIA [20], [21] is probably the most relevant technique from this category. SOFIA does not only encrypt the program, but prevents the execution of any tampered instruction via MAC checks. Unfortunately, this is rather costly. The average overhead of the PRINCE-based SOFIA implementation with 64-bit tags in terms of clock cycles and code size is 141 % and 203 %, respectively. On the other hand, an AEE-Light implementation with 64-bit sponge capacity is expected to yield less overheads, i.e., roughly 20 % and 40 % for runtime and code size, respectively. Note that, considering that the number and position of patches does not change when the sponge state/patch size is increased, extrapolating our results for a given program provides an exact overhead number for the program size and a plausible estimate for the performance overhead.

Another common approach to enforce CFI is to augment the processor with hardware monitors [7], [33], [43]. Typically, these monitors continuously check that the processor behaves as expected and raise an alert when an error is observed. The disadvantage of this approach is that deciding between correct and incorrect behavior (1-bit of information) effectively introduces a single point of failure for the error detection. As a result, implementing a reliable monitor becomes a challenge on its own. Additionally, these techniques can only provide integrity/authenticity and do not offer confidentiality.

Trusted Execution Environment. TEEs typically also provide confidentiality and authenticity for code. However, TEEs operate on a completely different level of granularity than SCFP. The authenticated encryption in Intel SGX [34], for example, only ensures that the code and data in memory is protected against tampering. Physical faults in caches or on processor internal buses, on the other hand, are still possible. Also, SGX does not prevent common software attack techniques like code-reuse attacks within enclaves. Therefore, additional CFI schemes are needed to reach similar properties as SCFP for code.

7. Conclusion

IoT devices are exposed to a wide range of attacks, such as code injection, code-reuse attacks, fault attacks, and IP theft. While there are suitable countermeasures for each of these attacks, nowadays' IoT devices hardly implement any protection mechanism. On the other hand, it requires several of the existing countermeasures to mitigate all of the mentioned attacks. However, a combination of different countermeasures is hard to analyze and may result in overheads that are too large for IoT devices.

To overcome this limitation, this work introduced Sponge-based Control-Flow Protection (SCFP). SCFP uses sponge-based authenticated encryption to encrypt and authenticate software with instruction-level granularity. During runtime, a hardware extension continuously decrypts instructions at the latest possible point before the processor's decode stage. As a result, SCFP effectively protects confidentiality and authenticity of the software IP, and provides fine-grained CFI to prevent code injection, code reuse, and control-flow fault attacks on the processor chip. The CFI enforced by SCFP is compatible with interrupts and standard operating systems. To emphasize the flexibility of SCFP, we further introduced three different instances of SCFP for different application purposes. While AEE provides all security features at cryptographic levels of security, AEE-Light reduces the level of software authenticity in trade for smaller memory overhead. In addition, IE is a very lightweight CFI scheme without any guarantees w.r.t. software authenticity and confidentiality. Finally, we demonstrated the practicality of SCFP by extending a RISC-V processor core with an instance of AEE-Light and evaluating a set of benchmarks. Our evaluations indicate that AEE-Light is suitable for many IoT scenarios with low code size and runtime overheads of 19.8 % and 9.1 % on average, respectively.

Acknowledgements

We thank Florian Mendel and Christoph Dobraunig for their helpful comments and many fruitful discussions. Furthermore, we thank Frank K. Gürkaynak, Germain Haugou, Beat Muheim, and Prof. Luca Benini from the Institute for Integrated Systems (IIS) at ETH Zurich for supervising David Schaffenrath during his time at ETH Zurich and for manufacturing the ASIC.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

erc European Research Council Established by the European Commission

References

[1] "Mirai botnet," 2016. [Online]. Available: http://github.com/jgamblin/Mirai-Source-Code

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow Integrity Principles, Implementations, and Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 4:1–4:40, Nov. 2009.

[3] E. Andreeva, B. Bilgin, A. Bogdanov, A. Luykx, F. Mendel, B. Mennink, N. Mouha, Q. Wang, and K. Yasuda, "PRIMATEs v1.02 Submission to the CAESAR Competition," Sep. 2014. [Online]. Available: http://primates.ae/

[4] E. Andreeva, B. Bilgin, A. Bogdanov, A. Luykx, B. Mennink, N. Mouha, and K. Yasuda, "APE: Authenticated Permutation-Based Encryption for Lightweight Cryptography," in *Fast Software Encryption*, ser. LNCS. Springer Berlin Heidelberg, Mar. 2014, no. 8540, pp. 168–186.

[5] E. Andreeva, J. Daemen, B. Mennink, and G. V. Assche, "Security of Keyed Sponge Constructions Using a Modular Proof Approach," in *Fast Software Encryption*, ser. LNCS. Springer Berlin Heidelberg, 2015, vol. 9054, pp. 364–384.

[6] Anton Cherepanov, ESET, "Win32/industroyer: A new threat for industrial control systems," 2017. [Online]. Available: https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32_Industroyer.pdf

[7] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Trans. VLSI Syst.*, vol. 14, no. 12, pp. 1295–1308, 2006. [Online]. Available: https://doi.org/10.1109/TVLSI.2006.887799

[8] J.-P. Aumasson, P. Jovanovic, and S. Neves, "NORX v1," Mar. 2014.

[9] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," 2004, IACR Cryptology ePrint Archive, Report 2004/100.

[10] D. J. Bernstein, "CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness," Jan. 2016. [Online]. Available: http://competitions.cr.yp.to/caesar.html

[11] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Duplexing the sponge: single-pass authenticated encryption and other applications," 2011, cryptology ePrint Archive, Report 2011/499.

[12] ——, "On the security of the keyed sponge construction," *SKEW*, 2011. [Online]. Available: http://sponge.noekeon.org/SpongeKeyed.pdf

[13] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, "CAESAR submission: KETJE v1," Mar. 2014. [Online]. Available: http://ketje.noekeon.org/

[14] ——, "CAESAR submission: KEYAK v2," Dec. 2015. [Online]. Available: http://keyak.noekeon.org/

[15] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented Programming: A New Class of Code-reuse Attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 30–40.

[16] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçin, "PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract," in *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, ser. Lecture Notes in Computer Science, X. Wang and K. Sako, Eds., vol. 7658. Springer, 2012, pp. 208–225. [Online]. Available: https://doi.org/10.1007/978-3-642-34961-4_14

[17] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 161–176.

[18] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting bare-metal embedded systems with privilege overlays," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 289–303. [Online]. Available: https://doi.org/10.1109/SP.2017.37

[19] C. Cowan, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*, A. D. Rubin, Ed. USENIX Association, 1998. [Online]. Available: https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention

[20] R. de Clercq, J. Götzfried, D. Übler, P. Maene, and I. Verbauwhede, "SOFIA: software and control flow integrity architecture," *Computers & Security*, vol. 68, pp. 16–35, 2017. [Online]. Available: https://doi.org/10.1016/j.cose.2017.03.013

[21] R. de Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. D. Bosschere, B. Preneel, B. D. Sutter, and I. Verbauwhede, "SOFIA: software and control flow integrity architecture," in *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, L. Fanucci and J. Teich, Eds. IEEE, 2016, pp. 1172–1177. [Online]. Available: http://ieeexplore.ieee.org/document/7459489/

[22] R. de Clercq and I. Verbauwhede, "A survey of hardware-based control flow integrity (CFI)," *CoRR*, vol. abs/1706.07257, 2017. [Online]. Available: http://arxiv.org/abs/1706.07257

[23] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "ASCON v1.1 Submission to the CAESAR Competition," Aug. 2015. [Online]. Available: http://ascon.iaik.tugraz.at/

[24] ETH Zurich, "Pulpino source repository," 2017. [Online]. Available: https://github.com/pulp-platform/pulpino

[25] N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier," *White paper, Symantec Corp., Security Response*, vol. 5, no. 6, 2011.

[26] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*, ser. SecuCode '09. New York, NY, USA: ACM, 2009, pp. 19–26. [Online]. Available: http://doi.acm.org/10.1145/1655077.1655083

[27] Free60.org, 2012. [Online]. Available: http://free60.org/wiki/Reset_Glitch_Hack

[28] D. Gligoroski, H. Mihajloska, S. Samardjiska, H. Jacobsen, M. El-Hadedy, R. E. Jensen, and D. Otte, "π–Cipher v2.0," Aug. 2015.

[29] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," *arXiv:1507.06955 [cs]*, Jul. 2015.

[30] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, Jun. 2014, pp. 361–372.

[31] T. Korak and M. Höfler, "On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms," in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Sep. 2014, pp. 8–17.

[32] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, J. Flinn and H. Levy, Eds. USENIX Association, 2014, pp. 147–163. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[33] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," *IEEE Trans. Computers*, vol. 59, no. 6, pp. 847–854, 2010. [Online]. Available: https://doi.org/10.1109/TC.2010.32

[34] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013, pp. 1–1. [Online]. Available: https://software.intel.com/en-us/articles/innovative-instructions-and-software-model-for-isolated-execution

[35] B. Mennink, R. Reyhanitabar, and D. Vizár, "Security of Full-State Keyed and Duplex Sponge: Applications to Authenticated Encryption," 2015, cryptology ePrint Archive, Report 2015/541.

[36] P. Morawiecki, K. Gaj, E. Homsirikamol, K. Matusiewicz, J. Pieprzyk, M. Rogawski, M. Srebrny, and M. Wojcik, "ICEPOLE v1," Mar. 2014.

[37] M.-J. O. Saarinen and B. B. Brumley, "STRIBOBr2: "WHIRLBOB" Second Round CAESAR Algorithm Tweak Specification," Sep. 2015. [Online]. Available: http://www.stribob.com/

[38] Y. Sasaki and K. Yasuda, "How to Incorporate Associated Data in Sponge-Based Authenticated Encryption," in *Topics in Cryptology - CT-RSA 2015*, ser. LNCS, vol. 9048. San Francisco, CA, USA: Springer International Publishing, 2015, pp. 353–370.

[39] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561.

[40] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, V. Atluri, B. Pfitzmann, and P. D. McDaniel, Eds. ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[41] P. Team, "Pax address space layout randomization (aslr)," 2003.

[42] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2," EECS Department, University of California, Berkeley, Tech. Rep., May 2017. [Online]. Available: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[43] M. Werner, E. Wenger, and S. Mangard, "Protecting the control flow of embedded processors against fault attacks," in *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, ser. Lecture Notes in Computer Science, N. Homma and M. Medwed, Eds., vol. 9514. Springer, 2015, pp. 161–176. [Online]. Available: https://doi.org/10.1007/978-3-319-31271-2_10

[44] K. D. Wilken and J. P. Shen, "Continuous signature monitoring: low-cost concurrent detection of processor control errors," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 9, no. 6, pp. 629–641, 1990. [Online]. Available: http://dx.doi.org/10.1109/43.55193