Scrambling Compiler: Automated and Unified Countermeasure for Profiled and Non-Profiled Side Channel Attacks

Gabriele Magnani[1][0000−0001−9729−5826], Isabella Piacentini[1][1111−2222−3333−4444], Giovanni Agosta[1][0000−0002−0255−4475], Alessandro Barenghi[1][0000−0003−0840−6358], and Gerardo Pelosi[1][0000−0002−3812−5429]

Politecnico di Milano
Department of Electronics, Information and Bioengineering (DEIB)
name.surname@polimi.it

Abstract. Side channel attacks extracting secrets carried by the power consumption variations or electromagnetic emissions in embedded devices are a consolidated threat to the security of edge computing systems. Such attacks either employ a synthetic model for the device behaviour to predict secret-dependent components of the measured power consumption (non-profiled attacks), or obtain such a model in a data-driven fashion (profiled attacks). Protections against both profiled and non-profiled attacks are characterized by a significant overhead, typically one or two orders of magnitude in computation time, and a comparatively significant engineering effort to deploy them. Furthermore, such protections are designed to hinder one of the two aforementioned attack strategies. In this work, we propose a compiler-based methodology to automate the application of a comparatively inexpensive countermeasure able to hinder both profiled and unprofiled attacks. We experimentally validate our approach employing the AES symmetric cipher as our case study, and a Cortex-M4 based microcontroller as the target device. Our solution increases the Measurements-to-Disclose security metric by at least 5000× in an attacker-optimal scenario, and proves to be immune to Bayesian template- and SVM-based profiled attacks.

Keywords: Embedded Systems Security · Side-Channel Attacks · Automated Countermeasure Instantiation

1 Introduction

The pervasive nature of embedded computers has led to a variety of devices, which are both interconnected and process data requiring security guarantees. One of the main threats to the cryptographic means employed to provide such security guarantees is represented by the so-called Side Channel Attacks (SCAs). SCAs allow to sidestep the computational hardness of the problems on which modern cryptographic primitives are built as they obtain information on values

intended to be secret in the mathematical design of the primitive (e.g., symmetric keys, private signature keys). SCAs exploit the possibility to collect measures of physical properties, e.g., electromagnetic emissions, power consumption, or computation latency that are correlated to the secret data involved in the execution of the cryptographic primitives [24, 25, 31, 33]. The information can be gathered either simply leveraging not-invasive observations of the system working as intended (passive SCAs), or observing the behaviour of the system under induced faults (active SCAs, also known as fault attacks). To extract information from the side channel measurements, the attacker devises a model of the power consumption of the device when an operation of the executed algorithm is performed. Such an operation is chosen in such a way that it takes as operand an (unknown) value depending from a small portion of the secret to be retrieved. The attacker then proceeds to determine the values of the model for all possible values of the secret and proceeds determining which of them best fits the measured behaviour of the device, in turn identifying the value of the portion of the secret she wants to retrieve. The model construction can be performed in a purely data-driven way (the case of *profiled* SCAs) or employing a synthetic model of the computing platform (*non-profiled* SCAs) [25]. For profiled SCAs, a large variety of statistical methods have been explored, from straightforward, but information theoretically optimal Bayes classifiers [12], to Support Vector Machines (SVMs) [23], and neural networks [15].

While a variety of SCA countermeasures are available to prevent non-profiled SCAs [2,25], there is experimental evidence that profiled SCAs can be performed even in the presence of such countermeasures [10]. *Scramble Suit* [7], on the other hand, is a dedicated countermeasure to profiled attacks designed for hardware accelerators implementing a specific cryptographic primitive. Essentially, it aims at differentiating the behaviour of each individual device, by inserting computation dependent on the secret values, the inputs and a device unique fingerprint. The differentiation is obtained duplicating the component computing the cryptographic primitive, and having the duplicate act on the same inputs as the original component, but employing a different, device dependent key. Such an approach prevents an attacker from separating the contribution of the differentiation element to the side channel behavior from the one related to the secret she's trying to extract.

The Scramble Suit approach is, however, is vulnerable to non-profiled attacks. Indeed, a non-profiled attack can find out more than one fitting model to the device behaviour, and hence more than one admissible secret value, the attacker could simply try all the extracted secret values, at the cost of a relatively small amount of exhaustive search. *Computation Interleaving (CI)* was proposed in [28] as a unified countermeasure for both classes of attacks, once again targeting hardware implementations. CI relies on employing a single datapath to compute two, or more, instances of the cryptographic primitive to be protected, interleaving them temporally. In line of principle, temporal interleaving is replicable also in software implementations; however, the extremely high engineering cost of writing a tailored assembly-level algorithm, which is required

to have the needed fine-grained control of the dataflow in a CPU pipeline, makes its transposition impractical without resorting to any automation mechanism.

Contribution. In this work, we propose a method to automate the application of the Computation Interleaving approach to software realizations of cryptographic primitives. Our approach relies on a modified compiler pipeline, which transforms an unprotected implementation of a cryptographic primitive into one protected with the computation interleaving countermeasure proposed by [28], without requiring human intervention. The proposed *CI hardening compiler* removes the demand for a per-platform engineering effort, enabling the widespread application of the approach on commodity off-the-shelf embedded platforms. Our compiler transformation takes care of tracking CPU register reuse and pipeline dataflow across the entire code being emitted, a difficulty which arises from transposing into a software realization the CI countermeasure, because dedicated hardware components are not available. Indeed, since the information leakage in side channels stems from the procedure employed to compute a value, and hence the hardware computing it, the implicit sharing of hardware resource that takes place in software implementations (e.g., register reuse) has long been known to be a source of vulnerabilities, which are challenging and time consuming to manage manually [5, 11, 32]. Our approach completely removes the need for specific expertise and additional engineering effort when it comes to secure a software implementation of a cipher and leverages the capabilities of a compiler infrastructure to obtain precise dataflow tracking and emit the protected assembly implementation, while fulfilling the requirements for the CI countermeasure. We validate our approach on a commercial-off-the-shelf Cortex-M4 microcontroller platform, performing profiled and non-profiled attacks and making our side channel measurements in a setting ideal for the attacker, i.e., on the ChipWhisperer [27] open evaluation platform for SCAs. We decided to take as our case study the Thumb-2 instruction set architecture (ISA) as it is supported by all ARM Cortex-M cores, which are optimized for low-cost and energy-efficient integrated circuits, and are embedded in billions of consumer devices.

Related work. Side channel attacks and the related topics of detection of side channel vulnerabilities, countermeasure design, and their automated instantiation occupy a prominent place in the field of computer security [18]. As a recent systematic review reports [18], hardware and embedded systems security has steadily grown into the predominant topic in vulnerability analysis, during the course of the last decade. As such, a vast body of recent literature deals with the aforementioned problem, the coverage of which far exceeds the scope of this paper. We refer the reader to recent surveys for an overview of the various subtopics [20, 24, 33]. Providing automated instantiations of SCA countermeasures started with early attempts which employed a domain specific language or performed ad-hoc lexical analyses [6, 9]. Employing dataflow analysis techniques from the domain of compiler engineering allowed the authors of [1] to automatically delineate the attack surface for power consumption based SCAs against software implementations of ciphers, and automatically apply masking countermeasures, tailoring them to avoid unneeded computational overheads. Similar

approaches were also employed to prevent timing attacks [13, 36]. Finally, the systematic code analysis and transformation capabilities of a compiler framework were also employed to introduce a new approach to counteracting SCAs that relies on modifying the runtime computation of a cryptographic primitive, while preserving the semantic equivalence of the corresponding machine code, an approach known as *code morphing* [2, 3]. We advance the state of the art in the automated side channel attack countermeasure application by employing the code analysis and transformation capabilities of an industrial grade compiler, namely LLVM [21], to overcome the technical and performance limitations arising from a manual adaptation of the principles of the CI countermeasure to a software implementation of a cryptographic primitive. In particular, we leverage the compiler infrastructure to obtain systematic, instruction-level accurate scheduling, which prevents information leakage due to the sharing of the same microarchitectural components by the interleaved computations.

2 Background

In this section, we summarize the background on the Computation Interleaving countermeasure approach, and provide background notions on the structure of a compiler. Finally, we briefly describe the IT ARM instruction, available in the ARMv7 architecture, which we employ, in our chosen case study platform, to further enhance the efficiency of our approach.

2.1 Computation Interleaving Countermeasure

The Computation Interleaving (CI) countermeasure [28], employs the temporal interleaving of two (or more) computations of a block cipher as the means to introduce algorithmic noise to counteract both profiled and non-profiled side channel attacks. All the multiple computations of the same cipher are acting on the actual input of the primitive (in the following, a block cipher encryption for the sake of exposition clarity), while only one is employing the actual secret key. All other cipher instances employ different secret key values obtained combining a device fingerprint and the actual secret key so that the device keys are computationally indistinguishable from randomly drawn ones, and, knowing the actual secret key it is not possible to derive the device dependent keys. In [28] the authors propose either the use of a Physically Unclonable Function to derive the device dependent keys, or to encrypt the secret key under the device fingerprint as a key, assuming to have a dedicated encryption core to do so.

In the following, we thus consider a CI protected symmetric block cipher $\textsc{Enc}(k, p)$, where k is the secret key and p the plaintext, and consider a single, device dependent, *fake* key k^f. The computation of $\textsc{Enc}(k, p)$ must be interleaved with the one of $\textsc{Enc}(k, p)$ in such a fashion that the three properties reported in [28] are guaranteed. To describe these properties, we will denote with $S_{i,k}$ the state of the block cipher before the i-th round computation, and therefore $S_{0,k}$ will represent the plaintext and $S_{n,k}$ as the ciphertext of an encryption

procedure that applies consecutively n round computations. To realize the CI countermeasure the implementation must provide the following properties:

Property 1 (Execution Order Randomization) *The order of the operations on each pair of $S_{i,k}$ and S_{i,k^f} must be randomized.*

This property prevents the attacker from simply separating the true and fake computations by considering the time instant in which each one is being done by the device implementing the cryptosystem. Indeed, if the order in which the two cipher instances are computed is fixed, it is sufficient for an attacker to discard the computations over S_{i,k^f}, or employ the information deterministically provided by them for higher order side channel attacks [26].

Property 2 (No Same-Computation-State Overwriting) *A register should never store one after the other values from two subsequent states of the same computation, $S_{i,k}$ and $S_{i+1,k}$.*

Violating Property 2 would immediately imply that the side-channel behaviour of the implementation in the time instant when the value from $S_{i+1,k}$ overwrites the one from $S_{i,k}$ matches the one of an unprotected implementation. Indeed, such an overwriting action implies that the memory storing the values has a switching activity which is proportional to the Boolean difference, or xor the values being stored. Such a switching activity can be uniquely determined from a portion of the computation of the unprotected cipher acting on the correct secret key alone. Enforcing Property 2, instead, ensures that the value transitions in the memory elements have a side channel behavior which always depends on both the state of the cipher computing on the actual key, and the one of the cipher employing the fake key. This prevents an attacker from employing device model obtained via a data-driven strategy on a given device instance on a different one: their device keys will be different, in turn resulting in different side channel behaviours, in turn preventing a profiled attack from being effective. Similarly, the dependence of the side channel behavior on both the correct and the fake key, and the dependence of the fake key on the entire correct key and device key prevents an attacker from modeling synthetically the behavior of the device to lead a non-profiled side channel attack, without requiring to guess both the user and device keys altogether, or trying to single out the contribution of the fake key (which is unfeasible if Property 1 holds).

Property 3 (No Consecutive-Computation-Operations) *The same computational operation should not be performed twice in a row on subsequent states of the same computation, $S_{i,k}$ and $S_{i+1,k}$ by the same combinatorial logic.*

Not implementing this property would enable a first-order non-profiled attack from the switching activity of the combinatorial logic, following the same line of reasoning of a violation of Property 2. Indeed, while modeling the transitions of a given arbitrary combinatorial cone is more challenging, the amount of information required to do so is the same as for memories, i.e., knowing the sequence of their inputs.

Ensuring that the three properties hold in a dedicated hardware design is relatively straightforward, as the designer has full control over the dataflow and may add dedicated memories to avoid unwanted overwriting actions. By contrast, ensuring that a CPU executing a software implementation of a cipher does not violate any of them requires precise and systematic control over the instruction scheduling and register allocation. In this work, we consider in-order, single issue CPUs, such as the ones in the overwhelming majority of microcontroller-oriented designs (e.g ARM Cortex-M series), where it is possible to enforce such constraints from essentially an architectural point of view. Taking into account the effects of advanced microarchitectures, performing, e.g., dual issuing, requires to further specialize code emission per single microarchitecture. While this is feasible with a compiler based approach, including information on the microarchitectural leakage such as the ones from the analyses in [8] is beyond the scope of this work.

2.2 Preliminary Notions on Compiler Structure

We now provide a summary of the structure of a modern compiler pipeline, taking the LLVM compiler pipeline as an example and define the concepts from compiler theory which we will need in the following. The LLVM compiler infrastructure [21] works as a pipeline, where code, represented in the LLVM-IR format, is analysed and transformed through a sequence of separate steps called *passes*. The compiler is logically split onto three portions a *front-end*, a *middle-end* and a *back-end*. The front-end performs the parsing of the source code and produces an Intermediate Representation (IR) of it. The middle-end is composed of those passes that take as input and produce as output the LLVM-IR itself, thus remaining independent from both source and target languages. All the target machine-independent code optimization takes place there. Finally, the back-end performs machine-dependent optimization and produces the machine code. This modular and flexible infrastructure has made LLVM a trusted and popular choice for research and production compilers.

The LLVM-IR represents the source as a set of functions, each one of which has its code as a straight-line sequence of IR instructions, also known as *Basic Blocks* (BBs), delimited by branch IR instructions. Each instruction i is a tuple containing the instruction opcode (e.g., `add`) and its operands in the form of virtual registers. Virtual registers are meant to abstract from the actual number of available registers in the target architecture, and allow to represent the IR following the Static Single Assignment (SSA) principle [30]. An IR in SSA form is such that every instruction computing a new value sees the value stored into a new virtual register. It is praxis to say that an instruction *defines* its output virtual register, while *using* its inputs. We denote with $def(i)$ the set of virtual registers defined by the instruction i and with $use(i)$ the set of its used ones.

To perform automated analyses on the LLVM-IR of a program, it is commonplace to think of its instructions (or, if convenient its BBs), as logically arranged into a graph. Two graph representations are common: the *control flow graph* and the *data flow graph*. In the former, instructions represent nodes of

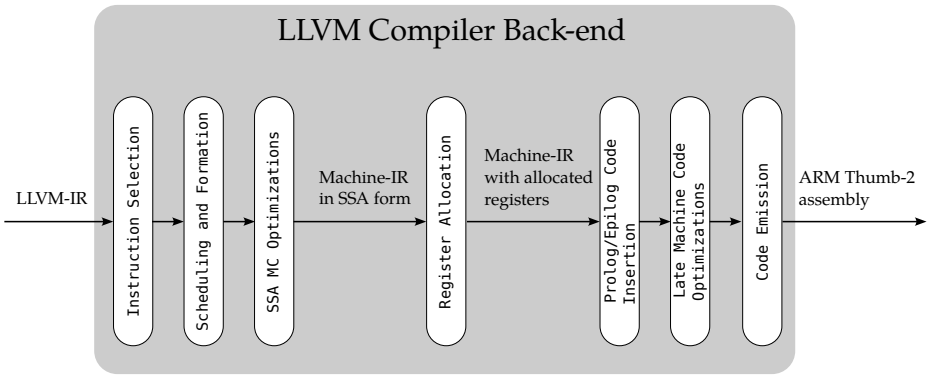

Fig. 1: High level overview of the LLVM compiler backend, highlighting the main sets of passes; the LLVM back-end

the graph, while arcs connect two instructions if the source appears right before the destination in program order. Note that this implies that conditional branch instructions are sources for more than one arc. In the data flow graph representation, instructions are represented with nodes, while the arcs have as source node the one defining a variable and as destination nodes all the instructions which use the said variable. The data flow graph thus provides a representation of which nodes, i.e., instructions, depend on another one, as the latter will compute at runtime the values which they will use as inputs.

We will, in our augmentations of the compiler pipeline, make use of the notion of *dominance*. An instruction i *dominates* another instruction j (and we denote it with $i \prec j$) when considering one of the two graph representations of the IR if all paths from the beginning of the function to j must pass through i. Furthermore, i *immediately dominates* j if it dominates it and no other instruction l dominates j, unless it also dominates i, that is $\forall l \in I, l \prec j \implies l \prec i$. The same notions of dominance and immediate dominance can be expressed for basic blocks rather than for individual instructions by simply coalescing together sequences of computational instructions uninterrupted by branches in the control flow graph. Being in need of controlling the scheduling of instructions and the register allocation, our modifications to the LLVM toolchain will mainly involve the LLVM backend, of which a logic view is represented in Figure 1. The LLVM backend performs the *Instruction Selection* pass first, which replaces the LLVM-IR instructions with instructions from the target machine Instruction Set Architecture (ISA). It is commonplace to refer to the resulting IR as Machine-IR, to highlight the fact that its form depends on the target machine. The resulting Machine-IR is considered in its graph logical organizations described before, and the *Scheduling and Formation* passes, perform a topological sort of the nodes of the graph themselves, yielding a sequence of target instructions. In doing so, the Scheduling pass aims at minimizing the amount of pipeline stalls, wherever

```
ITETT  EQ
mov r0, r4 # Then
mov r1, r5 # Else
mov r2, r6 # Then
mov r3, r7 # Then
```

Listing 1.1: This list illustrates an example of `IT` (If-Then) instruction. In this case, `<cond>` is set to `EQ` (equal). The conditional flags are defined as `ETT`. This means that if the first instruction executes successfully, both the third and fourth instructions will also be executed. If the first instruction does not execute, only the second instruction will run

possible. Machine-dependent optimizations that operate with a program representation in SSA form are then performed by the *SSA MC Optimizations* stage. At this point, the representation *exits* the SSA form via the *Register Allocation* pass, which maps the infinite virtual register file of the SSA form onto the physical register file. The *Prolog/Epilog* stage inserts the prolog and epilog code for each function, and then the *Late Machine Code Optimizations* are performed, which are optimizations that operate on the target instructions after the resolution of the register allocation, while the *Code Emission* stage emits the assembly code.

2.3 The ARM IT Instruction

Traditionally, the ARM architectures have always offered the possibility of computing *predicated instructions*, i.e., commit the result of an instruction if and only if a given flag, or set of flags is asserted in the program status word. Such a feature relied on a portion of the instruction encoding representing the encoding of the condition itself. Predicated instructions allow to essentially eliminate the cost of executing branch instructions in if-then-else conditional constructs. To retain this flexibility even in the compact encoding of the Thumb-2 encoded instruction sets (which are now used both in microcontroller grade and application grade ARM CPUs), the ARM Thumb-2 ISA includes an instruction known as the IT (If-Then) instruction. The IT instruction allows conditional execution of one to four instructions following it depending on a given condition code. IT instructions are also supported in the traditional ARM ISA as pseudo-instructions (and assembled into sequences of predicated instructions). The IT instruction works by specifying a condition (evaluating to a Boolean value) and a sequence of one to four markers stating if the following one to four instructions should be executed if the condition is true or false. This is encoded in assembly representing the IT instruction as `IT[x[y[z]]] <cond>` where `<cond>` serves as the condition for the initial instruction within the IT block, and `x`, `y`, and `z` represent the conditional flags for the subsequent instructions following the first one, with `T` (Then) and `E` (Else) being the available options. If an instruction is bound to a

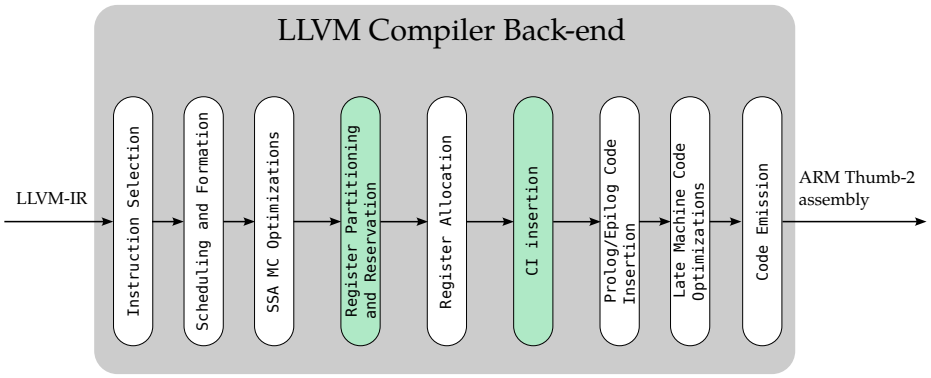

Fig. 2: LLVM pipeline modified to automate the insertion of the CI countermeasure: the additional passes are highlighted in teal

flag `T`, it will execute when `<cond>` evaluates to true. Conversely, an instruction with a flag E will execute if `<cond>` evaluates to false. Listing 1.1 is an example of an use of the IT instruction. In this instance, `<cond>` is set to EQ, i.e., the condition is true if result of the instruction preceding `ITETT` is equal to zero. The conditional flags are specified as ETT, which means that, if the first instruction executes, both the third and fourth instructions will also follow; otherwise, only the second instruction will be executed.

3 Scramble Hardening Compiler

In this section we detail how we designed a set of compiler passes which automatically realize the CI countermeasure in software, given an implementation of a cryptographic algorithm. A developer willing to employ our modified compiler pipeline only needs to provide the compiler with the name of the function implementing the cryptographic primitive to be protected and which parameter of the said function contains the secret key. This additional parameters are passed to the compiler as additional command line options. The function name is provided using the new command line option `scrambleName`. This option accepts a string input and is used to identify the function on which the transformation will be applied. It is not necessary to provide the mangled name of the function, as the scrambler will demangle the module's function name before making the comparison. Additionally, the index of the parameter containing the secret key is passed through the command line option `scrambleSecretKey`, which is specified as an enumerator in the argument list of the function. So in the case of the function `aes_sbox_encrypt(uint8_t rk[RK_SIZE], uint8_t output[ABLOCK_SIZE], uint8_t input[BLOCK_SIZE])` the compiler requires two additional command line parameters *-scrambleName=aes_sbox_encrypt -scrambleSecretKey=0*. As the CI insertion passes need precise knowledge about

```
eor  r3, r3, r4          eor  r9, r3, r4
strb r3, [sp, #12]       strb r9, [sp, #12]

b        .LBB1_4         b        .LBB1_4
.LBB1_4:                 .LBB1_4:
```

Listing 1.2: This listing demonstrates the impact of the CI insertion pass on an instruction that uses and defines the same register. The original instruction is shown on the left, with the use-def chain of `r3` highlighted and underlined in red. On the right, the instruction has been modified to use a new register, `r9`, and this change has been propagated to all subsequent uses

the register usage, as well as the arrangement in memory of the machine instructions, it cannot be implemented within the middle-end where such information is unavailable. Instead, it must be implemented in a target-aware way, within the LLVM back-end. To this end, we modified the LLVM compiler pipeline as represented in Figure 2.

While providing the three properties described in Section 2 is easiest tackling them in their enumeration order when done in a hardware realization, doing so through code transformations needs to first tackle the requirement of non-memory element sharing imposed by Property 2, acting on the register allocation strategies of the compiler. Considering the LLVM back-end pipeline depicted in Figure 2, we thus need to act before and after the *Register Allocation* pass. In particular, we split our operations in a *Register partitioning and reservation* pass, computed before the register allocation, and the *CI insertion* pass computed after.

3.1 Achieving Property 2 – avoiding register overwrite with the same computation

Achieving Property 2 requires that no value from a state $S_{i+1,k}$ is stored in the same register as the state from the same computation $S_{i,k}$. From a computation perspective, $S_{i,k}$ is represented by variables which are used by instructions computing $S_{i+1,k}$ (i.e. defining variables which represent it). Therefore, for Property 2 to hold, it must hold that:

$$\forall i \in I,\, def(i) \cap use(i) = \emptyset \tag{1}$$

that is, an instruction should not use (read) and define (write) the same register. This fact is usually not true, as instructions are allowed to employ the same architectural register as both input and output. Listing 1.2 shows an example of a basic block where instruction `eor r3, r3, r4` uses and defines the same

```
ldr  r5, [sp, #320]
```

```
ittee   eq
ldr  r2, [sp, #596]
ldr  r5, [sp, #320]
ldr  r5, [sp, #320]
ldr  r2, [sp, #596]
```

Listing 1.3: This listing shows the effects of the CI insertion on an instruction that uses memory. On the left, we have the original instruction in red underlined. On the right, the instruction was duplicated (blue dashed underlined) and cloned in reverse order. Additionally, the CI insertion has allocated a new stack space for the instruction

register `r3`, as well as the transformed code that the CI insertion generates. It can be noted that the definition of `r3` is replaced with a different register `r9`, and the subsequent use of `r3` in the next instruction is also replaced with a use of `r9`, preserving the original semantics. We note that, in case further uses of the replaced register, `r3` are present in BBs which may follow it in program order, there is also a need to move back the value onto `r3` before the branch leaving the current BB. To avoid violations of property 2, we adopt a *random precharging* approach, i.e. we overwrite `r3` right before inserting a move instruction which restores the value in `r3` itself from `r9`.

Register Set Partitioning. To automatically enforce this transformation, the CI inserting compiler passes needs to control how the registers are allocated in each instruction. CI insertion divides the registers into three disjoint sets: the allocation (R_A), instruction-remapper (R_I), and mirror (R_M) sets. The allocation set R_A contains the minimum number of registers necessary to carry out the *Register Allocation* pass of LLVM. The instruction-remapper set R_I contains the register that will be used to remap a register if it is used and defined by the same instruction. The CI register partitioning pass attempts to build an isomorphism $f_I : R_A \xrightarrow{\sim} R_I$ between the allocation and remapper sets if possible, as this guarantees that the remapping process can always be performed. If that is not possible, the cardinality of the remapper set will be less than the allocation set, but there will be a chance that the remapping step will fail. For the mirror set it is always defined an isomorphism $f_M : R_A \cup R_I \xrightarrow{\sim} R_M$ between all the already allocated registers and itself. Thus, $|R_A| + |R_I| = |R_M|$ is always true, and $|R_A| \geq |R_I|$. If $|R_A| = |R_I|$, the CI insertion is guaranteed to be possible. Since the *Register Allocation* requires at least 3 registers, a total of 12 general purpose registers are needed to ensure the protection of an arbitrary cipher function. This is sufficient for a wide range of general purpose RISC ISAs. In particular, when implementing AES on an ARM architecture, which is endowed with 13 general-purpose registers, the CI insertion will store three registers in the alloca-

tion and instruction-remapper set, while six registers are reserved in the mirror set. For architectures with very small register files, there is a possibility that the scrambling process cannot be successfully completed, which will be analyzed in the following.

The R_A, R_I, and R_M sets are constructed on a per-function basis and are populated with randomly chosen registers, so for each CI-protected function, the registers in each set will be uncorrelated. CI register partitioning forces the *Register Allocation* pass to only use registers available in the R_A set generated for the specific function.

Register Remapping. After *Register Allocation*, CI insertion scans the machine instructions and detects those that violate the condition expressed in Eq. (1). For each such instruction i that defines and uses register r, the CI insertion employs the instruction remapper set R_I to find an available remapping register r'. It then performs a remapping of the definition performed by i from r to the new register r'. Then, all the uses of r must be remapped to uses of r' along the paths in the code until r is redefined by a different instruction. Therefore the remapping might need to be propagated through different basic blocks. If a basic block BB_j has only one parent, and the remapped basic block BB_i dominates it immediately, the remapping can continue as if the instructions in BB_j belonged to BB_i (i.e., ignoring the branch), since no other definition of the same register can reach them. Otherwise, the CI insertion pass needs to insert a fragment of code that restores the remapped value from r' to the original register r. If the CI partitioning was able to create an isomorphism between the allocation and the instruction-remapper set, this process can be easily guaranteed to terminate. Indeed, every register in R_A has its own remapping register in R_I, and therefore the instruction that restores the original register r can be always performed, since the remapping register r' cannot have been overwritten except by another write to r in the original code, which however breaks the def-use chain, and therefore removes the need to restore the value of r. If this is not the case, the CI insertion keeps track of whether each register in R_I is currently in use, and attempts to use a free register when it is needed. If none can be found, then the modified LLVM compiler emits a warning as it is not possible to apply the CI insertion, given the resources at hand.

3.2 Adding the Fake Cipher Computation

A second step within the CI insertion is needed to address the insertion of the fake computation. This replica of the original cipher code must be interleaved with the original code. The availability of the mirror register set R_M ensures that the interleaving can be performed without affecting the semantics of the original (true) computation. Indeed, since $R_M \cap (R_A \cup R_I) = \emptyset$, the two computations are guaranteed not to interfere in terms of register usage.

Specifically, the CI insertion generates a new copy i^f of each instruction i in each basic block. Then, it performs a remapping of the registers used and defined

```
.LBB1_1:

movw    r7, #2056    @Only inserted
movt    r7, #20486   @when the random
ldr     r6, [r7]     @number needs
str     r6, [sp, #8] @to be reloaded

ldr     r7, [sp, #8]
tst     r7, #1
lsr     r7, r7, #1
str     r7, [sp, #8] @Optional
ittee   eq
...
```

Listing 1.4: This listing explains how the Scramble pass uses and reloads a random number. Red underlined instructions query the random number generator only if the stored number is depleted or at the beginning of a basic block with multiple parents. Blue dashed underlined instructions reload the random number and test if it is set. Orange wave underlined loads and stores the random number if the register is not cluttered

by i^f with the isomorphism f_M. While the generated pair of instructions do no share any register, $(def(i) \cup use(i)) \cap (def(i^f) \cup use(i^f)) = \emptyset$, memory locations for the accesses of the fake cipher instruction i^f also need to be allocated in a separate space. Listing 1.3 shows how the CI insertion handles an instruction that accesses memory by allocating a mirror stack region. Since memory space is not as scarce as registers, allocating this mirror stack region is trivial, requiring only simple bookkeeping. In this, we assume, for the sake of simplicity, that the entire cipher is inlined in a single function (as it is common to be the case for symmetric ciphers).

Randomizing the Ordering of True and Fake Instructions.
The last property that the CI insertion compiler needs to enforce is that true and fake instructions are randomly interleaved during the execution – i.e., at each execution of each pair of instruction, it should be randomically decided whether the true or fake instruction is executed first. Indeed, a static order of execution would allow a side-channel attacker to separate statically the fake computation from the true one by simple timing.

To this end, an architecture-independent approach is to generate, for each instruction-fake instruction pair the assembly semantical equivalent of an if-then-else construct, where the then-block and else-block are constituted by the two possible schedules of the instruction pairs. The condition of the if-then-else construct is then obtained drawing a random bit, therefore guaranteeing that the order in which an instruction and the corresponding fake one are executed is

chosen randomly. The aforementioned approach, which is architecture agnostic, however incurs in the overhead given by the insertion in the control flow of a non negligible amount of conditional branches. While this was shown to be acceptable in the context of the common overheads for side channel attack countermeasures [3], we chose to leverage the characteristics of the ARM Thumb-2 ISA to reduce the overhead to a minimum. To this end, we leverage the IT conditional block instructions. We recall that an IT instruction defines a block of one to four instructions that are executed conditionally, employing the same condition code that is read by the IT instruction itself. It is thus possible to employ an IT instruction driving the execution of the subsequent four computational instructions to our ends. To do so, Each instruction and its fake duplicate are emitted twice, the first time in a given order, while the second time in reverse order. The resulting four instructions sequence is then prefixed with an IT instruction. The condition of the IT block is set to execute the former pair of instructions if the condition is true, and the latter if the condition is false. Listing 1.3 shows how the CI insertion performs the cloning and wrapping, i.e., employing the ITTEE instruction which specialises IT with the correct conditions.

ITTEE employs the bits of the processor status word set by the previous instruction to drive the conditional execution. Therefore, to ensure the randomness of the selection, a random bit must be the result of the previous instruction. To enforce this, the CI insertion compiler employs a simple code fragment that requires a memory mapped true random number generator. We note that such a feature is common on microcontrollers [35], as a secure and fast TRNG is critical for key generation tasks which are typical of security oriented devices. Our approach considers a TRNG able to produce an architectural word (32 bit) of randomness per load operation. The snippet is inserted before the ITTEE block to read and test a single bit of the random filled register, which is filled loading from the address at which the true random number generator of the target ARM processor is mapped. CI insertion also loads a fresh random word after the current one is completely depleted or at the beginning of a basic block with multiple parents, where it is impossible to know a priori whether the current word is depleted or not. A stored random number is considered depleted if, from its loading, each of its bits was used to seed an IT block; this typically means reloading every 32 blocks. Since the TRNG is not guaranteed to provide fresh randomness at each clock cycle, but instead it requires a fixed amount of clock cycles to do so, all the basic blocks with an estimated cycle count less than the random number refresh time are padded with `nop` instructions to ensure getting a fresh random number each time. Listing 1.3 shows how CI insertion generates the appropriate code to handle the IT condition, including the (re)initialization of the random number. If a spare register is available (which is the case for the ARM register file), it can be used to reduce the number of load and store operations performed on the random number, by keeping it in a register.

These last passages fulfill the remaining required properties. In fact, even though this passage enforces randomization between the original instruction and a cloned one, the reloading of the random bit before each IT instruction is enough

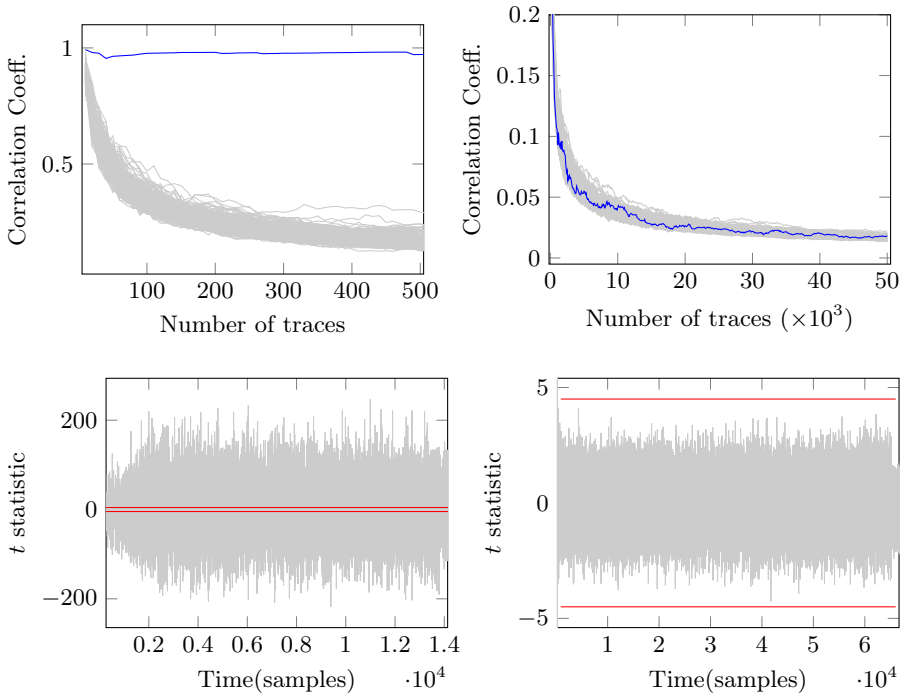

Fig. 3: Non-profiled CPA against an unprotected (top left) and protected implementation (top right): correlation coefficient as a function of the number of traces, correct key hypothesis in blue. Results of the non-specific t-test for the unprotected (500 traces, bottom left) and protected (50k traces, bottom right) implementation. Threshold not to be exceeded (± 4.5) for 99.999 confidence in protection in red

to ensure that no combinatorial Boolean function in the datapath is driven by two subsequent values of the same computation in two consecutive clock cycles.

4 Experimental Evaluation

In this section, we report the results of our experimental validation of the effectiveness of the proposed countermeasure in hindering both profiled and unprofiled attacks. Our benchmarking platform of choice is the ChipWhisperer 1200 Pro [27], a widely available hardware toolkit designed specifically to obtain high SNR measurements for side channel attacks, and provide a common ground for attack and countermeasure effectiveness comparisons. We employed a 32-bit STM32F4 ARM Cortex-M4 microcontroller. The same setup was employed to measure both the unprotected code, and the two instances of virtual devices having their power consumption profile altered with the scrambling countermeasure.

Sampling occurred synchronously with the target device clock signal, providing one sample per clock cycle. This clock-locking approach eliminates artifacts caused by potential clock jitter and was facilitated by the ChipWhisperer acquisition feature, which synchronizes the target device clock with an external crystal oscillating at 7.37 MHz. This in turn implies that no portability issues in the models are coming from trace misalignments, clock jitter, or process variability across devices, leaving only our scrambling countermeasure as a defense against profiled attacks. As our case study, we chose a reference implementation of the AES-128 block cipher [34] employing a single S-Box to compute the SubBytes primitive. From a performance standpoint, we report that the Scramble hardened code runs the entire AES cipher in 64.9k clock cycles of the target platform, while the unprotected AES implementation takes 13.8k clock cycles. The resulting slowdown ($\approx 4.6\times$) is compares quite favourably with common countermeasures tackling only unprofiled attacks such as masking, and is considered to be more than acceptable in practice. To put the figure in perspective, the state of the art, optimized masking schemes in [14] have the same AES run in 463k to 2.7M clock cycles on a 3.2 GHz Intel CPU.

Non-profiled Attack Security Validation. We validate the robustness of the proposed countermeasure in a threefold way: i) we perform a Correlation Power Analysis (CPA) attack, which is information theoretically optimal under the widely accepted assumption that the information leakage is proportional to the switching activity [16]; ii) a non-specific (i.e., model independent) t-test, as recommended by ISO/IEC 17825:2024 [19] and iii) the measured signal to noise (SNR) ratio w.r.t. the information which is expected to be leaked (i.e., Hamming distance between two intermediate values in a register). Figure 3 reports the results of both the CPA and the non-specific t-test. The results from the CPA, depicted in the two topmost figures show how Pearson's correlation coefficient for the correct key hypothesis (in blue), reaches a value close to the maximum possible (i.e., 1) after only 10 traces are employed for the attack in the unprotected device case (top left). We note that the Guessing Entropy (GE) score drops to its minimum consistently repeating the experiment 10 times. The protected design results (top right) show how employing $50k$ traces the correlation coefficient of the correct key hypothesis does not emerge w.r.t. the one of the wrong key hypotheses. This in turn implies an increase in the number of Measurements to Disclose (MtD) the secret value by at least even with $5000\times$, as no correct key retrieval is achieved even with all the $50k$ traces. The non-specific t-test results provide an analogous strong validation: indeed, the test on the unprotected implementation exceeds the threshold ± 4.5 for 99.999% confidence in the protection by more than 40 times when employing only 500 traces (bottom right). The protected implementation retains t statistic values well within the threshold, even when $100\times$ the traces are employed for the test. Finally, we report that the peak SNR (measured with $50k$ traces) for the information leakage on the AES SubBytes operation is 3.3, while the one on the protected implementation it is lowered to $2.9 \cdot 10^{-3}$.

Table 1: Classification accuracy of TAs and SVM on an unprotected (**u**) and protected implementation (**p**). Templates with 2k traces; attacks performed against 10k traces per device

Attack type	Feature reduction	Accuracy u	Accuracy p
Template	**NICV**	91.2%	50.4%
	SOST	98.8%	51.3%
	PCA	96.8%	50.2%
SVM	**NICV**	89.4%	50.1%
	SOST	94.5%	50.3%
	PCA	92.5%	49.9%

Profiled Attack Security Validation. We evaluate the resistance against profiled attacks employing the information theoretically optimal Bayesian Templates Attack (TAs) and Support Vector Machines (SVMs), which were shown to be an efficient alternative to TAs, especially if paired with an appropriate feature reduction approach [22]. We note that the use of machine learning techniques (e.g. neural network classifiers and deep learning methods) yields an efficiency advantage over the aforementioned TAs in cases where experimental noise, clock jitter or other disturbances are hindering the measurements. To the end of providing the most favourable scenario to the attacker, we took measures to minimize noise and remove the other hindrances from our attack evaluation, so that the efficiency advantage is not needed for the evaluation. Feature reduction is practically mandatory in our scenario, as the number of samples per trace (corresponding to the dimensionality of the random variable of which we need to estimate a distribution) is in the tens of thousands range. Employing raw traces would indeed fall into a *curse-of-dimensionality* issue for the classifier [29]. We employ three different feature reduction approaches: sample selection with the maximum SOST [17] or Normalized InterClass Variance (NICV) [4] score, and Principal Component Analysis (PCA) [22]. For the NICV score, we choose to classify the traces according to the same intermediate value hypothesis yielding a successful unprofiled attack, to avoid the computation of an interclass variance across poorly classified traces. For the PCA-based feature reductions, we choose to keep the 25 most informative dimensions of the projected space, as they were measure to contain 95% of the information.

For all device instances, we build two device profiles, selecting the first round key bit as the target of our attack, employing 1k traces to build each profile. Subsequently, we classify 3k traces (none in common with the set used in the modeling phase) coming from different device instances. Table 1 reports the outcome of the classifications made model TA results.

Profiled attacks succeed on the unprotected implementation with extremely high accuracy (94.5% for SVMs and 98.8 for TAs) given an appropriate choice of

the feature reduction strategy. accuracy against the unprotected device instance, while they obtain an accuracy in the 49.9%–51.3% range for all protected implementations, regardless of the feature reduction technique, or profiled attack approach. This provides strong evidence of the non-effectiveness of TAs against our countermeasure, as the fluctuations around the accuracy of a random guess are within a range of plausibility for statistical artifacts. We note that, in our scenario where single-bit (i.e., two-class) template attacks are performed, the Guessing Entropy and the accuracy scores coincide. Indeed, computing the GE in our template attacks (with the possible key ranks being only 0 and 1) yields 0.51 as the average rank, further pointing to the ineffectiveness of template attacks against the protection, in our evaluation scenario.

5 Concluding Remarks

In this work, we proposed method to perform the fully automated application of a Scramble hardening countermeasure to a software implementation of a symmetric cipher, by means of a modified compiler pipeline. Our approach allows to obtain a significant improvement in the resistence against non profiled side channel attacks ($\geq 5000\times$ the MtD) and hinders successfully profiled attacks. The proposed countermeasure compares favourably in terms of computational efficiency, as it is more efficient than state-of-the-art masking schemes, while providing protection from both unprofiled and profiled attacks.

Acknowledgments. This work was supported in part by project SERICS (PE00000014) under the Italian NRRP MUR program funded by the EU - NGEU.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Agosta, G., Barenghi, A., Pelosi, G.: A code morphing methodology to automate power analysis countermeasures. In: Groeneveld, P., Sciuto, D., Hassoun, S. (eds.) The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012. pp. 77–82. ACM (2012). https://doi.org/10.1145/2228360.2228376
2. Agosta, G., Barenghi, A., Pelosi, G.: Compiler-Based Techniques to Secure Cryptographic Embedded Software Against Side-Channel Attacks. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(8), 1550–1554 (2020). https://doi.org/10.1109/TCAD.2019.2912924
3. Agosta, G., Barenghi, A., Pelosi, G., Scandale, M.: The MEET approach: Securing cryptographic embedded software against side channel attacks. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **34**(8), 1320–1333 (2015). https://doi.org/10.1109/TCAD.2015.2430320
4. Archambeau, C., Peeters, E., Standaert, F., Quisquater, J.: Template Attacks in Principal Subspaces. In: Goubin, L., Matsui, M. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama,

Japan, October 10-13, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4249, pp. 1–14. Springer (2006). https://doi.org/10.1007/11894063_1
5. Balasch, J., Gierlichs, B., Grosso, V., Reparaz, O., Standaert, F.: On the Cost of Lazy Engineering for Masked Software Implementations. In: Joye, M., Moradi, A. (eds.) Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8968, pp. 64–81. Springer (2014). https://doi.org/10.1007/978-3-319-16763-3_5
6. Barbosa, M., Moss, A., Page, D.: Constructive and Destructive Use of Compilers in Elliptic Curve Cryptography. J. Cryptol. **22**(2), 259–281 (2009). https://doi.org/10.1007/S00145-008-9023-0
7. Barenghi, A., Fornaciari, W., Pelosi, G., Zoni, D.: Scramble Suit: A Profile Differentiation Countermeasure to Prevent Template Attacks. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(9), 1778–1791 (2020). https://doi.org/10.1109/TCAD.2019.2926389
8. Barenghi, A., Pelosi, G.: Side-channel security of superscalar CPUs: evaluating the impact of micro-architectural features. In: Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018. pp. 120:1–120:6. ACM (2018). https://doi.org/10.1145/3195970.3196112
9. Bayrak, A.G., Regazzoni, F., Brisk, P., Standaert, F., Ienne, P.: A first step towards automatic application of power analysis countermeasures. In: Stok, L., Dutt, N.D., Hassoun, S. (eds.) Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011. pp. 230–235. ACM (2011). https://doi.org/10.1145/2024724.2024778
10. Bronchain, O., Durvaux, F., Masure, L., Standaert, F.: Efficient Profiled Side-Channel Analysis of Masked Implementations, Extended. IEEE Trans. Inf. Forensics Secur. **17**, 574–584 (2022). https://doi.org/10.1109/TIFS.2022.3144871
11. Casalino, L., Belleville, N., Couroussé, D., Heydemann, K.: A Tale of Resilience: On the Practical Security of Masked Software Implementations. IEEE Access **11**, 84651–84669 (2023). https://doi.org/10.1109/ACCESS.2023.3298436
12. Chari, S., Rao, J.R., Rohatgi, P.: Template Attacks. In: Jr., B.S.K., Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers. Lecture Notes in Computer Science, vol. 2523, pp. 13–28. Springer (2002). https://doi.org/10.1007/3-540-36400-5_3
13. Cleemput, J.V., Coppens, B., Sutter, B.D.: Compiler mitigations for time attacks on modern x86 processors. ACM Trans. Archit. Code Optim. **8**(4), 23:1–23:20 (2012). https://doi.org/10.1145/2086696.2086702
14. Coron, J., Rondepierre, F., Zeitoun, R.: High Order Masking of Look-up Tables with Common Shares. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(1), 40–72 (2018). https://doi.org/10.13154/TCHES.V2018.I1.40-72
15. Das, D., Golder, A., Danial, J., Ghosh, S., Raychowdhury, A., Sen, S.: X-DeepSCA: Cross-Device Deep Learning Side Channel Attack. In: Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019. p. 134. ACM (2019). https://doi.org/10.1145/3316781.3317934
16. Durvaux, F., Standaert, F., Veyrat-Charvillon, N.: How to Certify the Leakage of a Chip? In: Nguyen, P.Q., Oswald, E. (eds.) Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8441, pp. 459–476. Springer (2014). https://doi.org/10.1007/978-3-642-55220-5_26

17. Gierlichs, B., Lemke-Rust, K., Paar, C.: Templates vs. Stochastic Methods. In: Goubin, L., Matsui, M. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4249, pp. 15–29. Springer (2006). https://doi.org/10.1007/11894063_2
18. Heiding, F., Katsikeas, S., Lagerström, R.: Research communities in cyber security vulnerability assessments: A comprehensive literature review. Computer Science Review **48** (2023). https://doi.org/10.1016/j.cosrev.2023.100551
19. ISO/IEC 17825:2024: Information technology - Security techniques - Testing methods for the mitigation of non-invasive attack classes against cryptographic modules. www.iso.org (2024)
20. Javeed, A., Yilmaz, C., Savas, E.: Microarchitectural Side-Channel Threats, Weaknesses and Mitigations: A Systematic Mapping Study. IEEE Access **11**, 48945–48976 (2023). https://doi.org/10.1109/ACCESS.2023.3275757
21. Lattner, C., Adve, V.S.: LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO) 2004), 20-24 March 2004, San Jose, CA, USA. pp. 75–88. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665
22. Lerman, L., Bontempi, G., Markowitch, O.: Power analysis attack: an approach based on machine learning. Int. J. Appl. Cryptogr. **3**(2), 97–115 (2014). https://doi.org/10.1504/IJACT.2014.062722
23. Lerman, L., Bontempi, G., Markowitch, O.: A machine learning approach against a masked AES - Reaching the limit of side-channel attacks with a learning model. J. Cryptogr. Eng. **5**(2), 123–139 (2015). https://doi.org/10.1007/S13389-014-0089-3
24. Lou, X., Zhang, T., Jiang, J., Zhang, Y.: A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography. ACM Comput. Surv. **54**(6), 122:1–122:37 (2022). https://doi.org/10.1145/3456629
25. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks - revealing the secrets of smart cards. Springer (2007)
26. Messerges, T.S., Dabbish, E.A., Sloan, R.H.: Power Analysis Attacks of Modular Exponentiation in Smartcards. In: Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1717. Springer (1999). https://doi.org/10.1007/3-540-48059-5_14
27. NewAE Technology Inc.: ChipWhisperer-Pro CW1200. https://rtfm.newae.com/Capture/ChipWhisperer-Pro/ (2024)
28. Piacentini, I., Barenghi, A., Pelosi, G.: A Non Profiled and Profiled Side Channel Attack Countermeasure through Computation Interleaving. In: 26th Euromicro Conference on Digital System Design, DSD 2023, Golem, Albania, September 6-8, 2023. pp. 718–725. IEEE (2023). https://doi.org/10.1109/DSD60849.2023.00103
29. Picek, S., Heuser, A., Jovic, A., Bhasin, S., Regazzoni, F.: The Curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019**(1), 209–237 (2019). https://doi.org/10.13154/TCHES.V2019.I1.209-237
30. Rastello, F., Tichadou, F.B.: SSA-based Compiler Design. Springer Nature (2022)
31. Ronen, E., Shamir, A., Weingarten, A., O'Flynn, C.: IoT Goes Nuclear: Creating a Zigbee Chain Reaction. IEEE Secur. Priv. **16**(1), 54–62 (2018). https://doi.org/10.1109/MSP.2018.1331033

32. Seuschek, H., Santis, F.D., Guillen, O.M.: Side-channel leakage aware instruction scheduling. In: Brorsson, M., Lu, Z., Agosta, G., Barenghi, A., Pelosi, G. (eds.) Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC 2017, Stockholm, Sweden, January 24, 2017. pp. 7–12. ACM (2017). https://doi.org/10.1145/3031836.3031838
33. Spreitzer, R., Moonsamy, V., Korak, T., Mangard, S.: Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices. IEEE Commun. Surv. Tutorials **20**(1), 465–488 (2018). https://doi.org/10.1109/COMST.2017.2779824
34. of Standards, N.I., Technology: FIPS PUB 197 - Advanced Encryption Standard (AES). https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf (2001)
35. STMicroelectronics: STM32F407/417 32 bit Cortex-M Microcontroller Reference Manual. https://www.st.com/en/microcontrollers-microprocessors/stm32f407-417/documentation.html (2025)
36. Wu, M., Guo, S., Schaumont, P., Wang, C.: Eliminating timing side-channel leaks using program repair. In: Tip, F., Bodden, E. (eds.) Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018. pp. 15–26. ACM (2018). https://doi.org/10.1145/3213846.3213851