In Proceedings of the First Italian Conference on Cybersecurity (ITASEC17), Venice, Italy.

An Enhanced Dataflow Analysis to Automatically Tailor Side Channel Attack Countermeasures to Software Block Ciphers

Alessandro Barenghi and Gerardo Pelosi

Politecnico di Milano, Department of Electronics, Information and Bioengineering – (DEIB), Italy
alessandro.barenghi@polimi.it,gerardo.pelosi@polimi.it

Abstract

Protecting software implementations of block ciphers from side channel attacks is a significant concern to realize secure embedded computation platforms. The relevance of the issue calls for the automation of the side channel vulnerability assessment of a block cipher implementation, and the automated application of provably secure defenses. The most recent methodology in the field is an application of a specialized data-flow analysis, performed by means of the LLVM compiler framework, detecting in the AES cipher the portions of the code amenable to key extraction via side channel analysis. The contribution of this work is an enhancement to the existing data-flow analysis which extending it to tackle any block cipher implemented in software. In particular, the extended strategy takes fully into account the data dependencies present in the key schedule of a block cipher, regardless of its complexity, to obtain consistently sound results. This paper details the analysis strategy and presents new results on the tailored application of power and electro-magnetic emission analysis countermeasures, evaluating the performances on both the ARM Cortex-M and the MIPS ISA. The experimental evaluation reports a case study on two block ciphers: the first designed to achieve a high security margin at a non-negligible computational cost, and a lightweight one. The results show that, when side-channel-protected implementations are considered, the high-security block cipher is indeed more efficient than the lightweight one.

1 Introduction

Embedded systems have a high platform diversification in terms of both the available computational resources on board, the actual instruction set architecture of the underlying computing core, and the availability of both volatile and non volatile memory. Such significant differences result in corresponding high engineering costs to realize functional, energy efficient and secure embedded systems. In particular, providing sound security guarantees in the embedded system world is a crucial issue, as they are increasingly in charge of critical tasks, and their expected long field life results in failures causing concrete problems over potentially a decade's worth of devices, especially in the Internet of Things and automotive environments [10]. Among the most prominent threats to the security of embedded devices relying on cryptographic techniques, *Side-Channel Attacks* (SCAs) have proven to be one of the most consolidated and effective techniques [15]. SCAs break correct, specifications abiding implementations, employing the information on the secret parameters, i.e., the cipher keys, which is unwillingly transmitted as a change in environmental parameters of the computing device such as its power consumption or radiated electromagnetic emissions. While sound countermeasure techniques against them are present and consolidated in open literature [8, 14], their application to an unprotected implementation is still largely performed by hand. Such an approach entails a significant effort, as the countermeasure techniques are applied either to the assembly code of a software implementation, or to the low level netlist representation of a soon to be implemented circuit. Recently, research efforts directed at automated, design time, application of countermeasures have shown the feasibility of systematically applying them to software implementations, employing the AES cipher as a case study [1–4].

Contributions. We propose an improvement over the current state of the art of compiler-based automated SCA countermeasure application. We provide accurate identification technique determining the most effective choice of the key material (i.e., the user key and the expanded key computed during the cipher *keyschedule*) to be retrieved when attacking via SCA a block cipher. Our contribution pushes forward the abilities of the automated analysis proposed in [1] examining new block ciphers, proposing an efficient alternative which does not rely on modifying the runtime executed code as proposed in [2–4] The relevance of the enhanced analysis is substantiated in an experimental evaluation applying tailored countermeasures. The results show that the protected implementation of a high-security, general purpose cipher achieves speedups in the $93\times$-$190\times$ range against the protected lightweight cipher.

2 Background

Given a block cipher implementation, the side channel attack (SCA) workflow is a *known plaintext attack*, where the adversary knows both the plaintext and the ciphertext, and aims at retrieving the secret key through examining multiple runs of the algorithm on different inputs [12]. The attacker knows all the details of the implementation of the cipher and measures either the power consumption or the electromagnetic emissions of the device to derive information regarding the secret key. The main strength of an SCA lies in considering the effect of the secret key bits on the computation separately instead of as a whole, leading to a significant lowering in the security margin of the primitive. A typical attack starts by choosing an intermediate value of the cipher depending on a small key portion (e.g., 8 bits) and a known quantity (e.g., the plaintext). The side channel is then measured during the aforementioned operation, for a large set of randomly distributed input values. Subsequently, the attacker predicts the outcome of the side channel measurement relying on the knowledge of the inputs and making an hypothesis for all the possible values of the secret key portion taken into account. This yields a set of hypotheses: one for each value possibly taken by the portion of the secret key under attack. Each hypothesis is compared with the measured side channel, through the use of a statistical test (e.g., the Pearson's correlation coefficient), revealing the actual value of the secret key portion, as the prediction depending on it will best fit the measurements. A direct consequence of this workflow is that the number of hypotheses to be made grows exponentially in the number of secret key bits involved in the side channel prediction.

Countermeasures aimed at protecting block cipher implementations can be classified as *hiding* [12], *masking* [11, 14] and *morphing* [2–4]. The *masking* countermeasures provide a formally proven lower bound on the computational effort required to the attacker to subvert the protection. They invalidate the correlation between the quantities employed to predict the power consumption and the actual values processed by the underlying device. The principle is to add one or more random values (a.k.a., *masks*) to every sensitive intermediate variable. Sensitive variables are the ones storing a value depending on a portion of the cipher key.

In a masked implementation, each sensitive intermediate value is represented as split in a number of shares, s, which are all needed for its reconstruction For example, s shares are obtained as $s-1$ random values and the `xor` combination of them with the original input. The target algorithm is modified to perform the entire computation on the set of share-split values recombining them only at the end.

The instantaneous power consumption is independent from the original (non-masked) value, as unpredictable random values are newly generated at each run of the cipher. Typically, masking techniques are categorized by the number of masks, $d{=}s{-}1$, employed for each sensitive value, which is known as the order of the masking. A d-th-order masking can be broken by a $(d{+}1)$-th-order attack through combining $d{+}1$ measurements of the computations of different shares during the same cipher execution.

Currently, state of the art masking schemes are the Ishai-Sahai-Wagner (ISW masking) [11] and Threshold Implementations (TIs) [14]. While the former has been proposed as a generic framework

to perform share-split computations of generic Boolean operators (`and`, `not` and `xor`), the latter was explicitly designed to be employed in dedicated hardware implementations, ensuring that also transitions stemming from signal glitches do not leak information. In [14] the ISW masking and the TIs were proven equivalently provably secure in the case of software implementations (with careful register reuse). The ISW masking scheme, employed to secure our case study ciphers, has an $O(s^2)$ computational complexity in protecting `and`s and `or`s, an $O(s)$ for `xor`s, and an $O(1)$ complexity for `not`s. In addition to the computation of Boolean functions, to protect actual implementations of block ciphers the share-splitting of `load` operations must be dealt with.

The only provably secure scheme providing a constructive framework to perform secure table lookups on share split values for any number of shares is described in [8], and relies on adding fresh randomness to the entire contents of the s-way split table at each `load`. The proposed algorithm has a computational complexity of $O(n^2)$, where n is the number of table elements. Finally, we recall that protecting with a provably secure SCA countermeasure only the portion of the block cipher where less than the entire key is required to compute an intermediate value was proven to have the same computational security margin as protecting the entire cipher [6]. The key intuition making the proof hold is that leading a SCA exploiting the leakage coming from a computation involving the entire cipher key is equivalent to sorting in order of likelihood of being correct the entire keyspace.

3 Vulnerability Assessment Analysis

In this section we provide the preliminary notions on data-flow analysis, and subsequently report the enhanced key material identification allowing the analysis in [1] to be applied to any block cipher.

Definition 3.1 (Control-Flow Graph)**.** *Given a program P as a sequence of statements, a Control-Flow Graph (CFG) is a directed graph $\mathcal{G}_{\mathtt{C}}(V, A_{\mathtt{C}})$ where each statement is a vertex in V, and the arcs are ordered pairs of vertexes $(v_1, v_2) \in A_{\mathtt{C}}$, indicating that $v_1 \in V$ precedes $v_2 \in V$ in program order.*

Definition 3.2 (Data-Flow Graph)**.** *Given a program P as a sequence of statements, a Data-Flow Graph (DFG) is a directed graph $\mathcal{G}_{\mathtt{D}}(V, A_{\mathtt{D}})$ where each statement is a vertex in V, and the arcs are ordered pairs of vertexes $(v_1, v_2) \in A_{\mathtt{D}}$, indicating that the statement in v_2 has as an operand (i.e.,* uses*) a value computed (i.e.,* defined*) by v_1.*

From now on, we will also indicate the elements of V as *nodes* and consider both $\mathcal{G}_{\mathtt{C}}(V, A_{\mathtt{C}})$ and $\mathcal{G}_{\mathtt{D}}(V, A_{\mathtt{D}})$ as augmented graphs containing two additional nodes, $v_{\mathtt{begin}}$, $v_{\mathtt{end}}$, such that $v_{\mathtt{begin}}$ acts as the immediate predecessor of all the nodes which do not have one, and $v_{\mathtt{end}}$ acts as the immediate successor of all the nodes without successors. Moreover we will also consider the nodes of the graphs to be composed of single operation statements. A static analysis of a program P, represented either as its CFG or DFG, is called *Data-Flow Analysis* (DFA) if it aims to describe a well-defined property on how P manipulates its data at runtime. Being a static technique a DFA will only be able to compute a conservative approximation of the said property as no knowledge on the actual runtime execution flow is available to it. More formally, the definition of a DFA is as follows.

Definition 3.3 (Data-flow Analysis)**.** *Let P be a program, represented as either $\mathcal{G}_{\mathtt{C}}(V, A_{\mathtt{C}})$ or $\mathcal{G}_{\mathtt{D}}(V, A_{\mathtt{D}})$. Let $\mathbf{BV}^n$ be an n-dimensional Boolean lattice, $n \geq 1$, endowed with a partial ordering relation closed with respect to two operations, named as* meet *($\sqcup$) and* join *($\sqcap$), which yield the minimum and maximum of any pair of elements in $\mathbf{BV}^n$. A Data-Flow Analysis decorates each node of $\mathcal{G}_{\mathtt{C}}(V, A_{\mathtt{C}})$ (resp. $\mathcal{G}_{\mathtt{D}}(V, A_{\mathtt{D}})$) with an element of $\mathbf{BV}^n$ and describes a well-defined property of P through computing the solution of a set of simultaneous equations over the elements of $\mathbf{BV}^n$.*

A practical example of DFA is the computation of a program property known as *reaching definitions*, i.e.: the identification of all the possible points on the CFG of the program, $\mathcal{G}_{\mathtt{C}}(V, A_{\mathtt{C}})$, where a variable visible from a program statement may have been defined. In this context, the semantics of an element of $\mathbf{BV}^n$ map each one of its Boolean components to a the definition point of a program variable on the CFG. The solution of the set of simultaneous equations checked by the DFA consists in the values in $\mathbf{BV}^n$ (one for each node of CFG), which are expected to have a bit set if the corresponding definition point is visible from the statement to which the element of $\mathbf{BV}^n$ at hand is bound.

Depending on the nature of the program property which is being computed, it is commonplace to categorize data-flow analyses into *forward* DFAs or *backward* DFAs. Furthermore, the aforementioned set of simultaneous equations is expressed using use of the concepts of In-Set and Out-set, which are formally defined as follows.

Definition 3.4 (In-Set, Out-Set)**.** *Given a CFG $\mathcal{G}_{\mathtt{C}}(V, A_{\mathtt{C}})$, $In(v)$ is defined for each statement $v{\in}V$ the input set as the collection of elements in $\mathbf{BV}^n$ associated to all immediate predecessors of v: $In(v){=}\{\ d_{v'}{\in}\mathbf{BV}^n \mid d_{v'}$ is bound to $v',\ v'{\in}pred(v)\ \}$. The output set of a statement v, denoted as $Out(v)$ is defined as $In(v){\cup}\{d_v\}$, where d_v is the element of $\mathbf{BV}^n$ bound to v.*

For each node v of the CFG (resp. the DFG) of a program, the data-flow equations describing the property at hand can be specified as follows:

$$Forward\begin{cases} Out(v) &= f_v(In(v)) \\ In(v) &= \bigsqcup_{p\in pred(v)} Out(p) \end{cases} \qquad Backward\begin{cases} In(v) &= f'_v(Out(v)) \\ Out(v) &= \bigsqcup_{s\in succ(v)} In(s) \end{cases}$$

where $pred(v)$ and $succ(v)$ denote the sets of immediate predecessors and the immediate successors of v, while $f_v(\cdot)$ and $f'_v(\cdot)$ are functions depending on the nature of the considered property which specify the local computations to be performed on the values in $\mathbf{BV}^n$ associated with $In(v)$ and $Out(v)$.

The program property is computed by the DFA through applying iteratively the data-flow equations to an initial assignment of the values $d_v{\in}\mathbf{BV}^n$ for each node v in $\mathcal{G}_{\mathtt{C}}(V, A_{\mathtt{C}})$ (resp. $\mathcal{G}_{\mathtt{D}}(V, A_{\mathtt{D}})$) until a fixed point is reached. Usually, a DFA solver implements an iterative visiting strategy of the CFG (resp. DFG) which keeps a worklist of nodes. Such a list is initialized with all $v{\in}V$ and, until it is empty, a node is evicted to apply the data-flow equations to its In-set and Out-set. Once done so, all the nodes having their In-set or Out-set modified as a result are added to the worklist. Sound data-flow analyses have a proof of termination which is provided in terms of the existence of a fixed point for each possible initial assignment to the program nodes.

3.1 Security oriented DFA

To define a resistance metric against SCA for each instruction of a block cipher, we consider the data dependencies holding between the user key loading instructions and the outputs of every other instruction of the program. We model these data dependencies binding to each output bit of an instruction as an element $\mathbf{b}{\in}\mathbf{BV}^{|k|}$, where $|k|$ is the size in bits of the user key of the block cipher. The result of our DFA sets the i-th component of $\mathbf{b}$ to one if the associated output bit of the instruction depends on the i-th user key bit. It is useful to group all the elements of $\mathbf{BV}^{|k|}$ bound to the output bits of a given node v of the CFG as a $|v| \times |k|$ Boolean matrix, where $|v|$ is the size (in bits) of the output of the node. The SCA resistance of an instruction is formalized as follows.

Definition 3.5 (SCA resistance)**.** *Let v be a CFG node to which is associated a $|v|{\times}|k|$ Boolean matrix and let* COUNT(j) *be the count of the components of the j-th row of the matrix set to one, i.e., the number of user key bits on which the j-th output bit of v depends on. The SCA resistance of v, $\mathtt{r}(v)$, is defined as* $\min_j(\text{COUNT}(j))$ *if* COUNT$(j){>}0$ *for at least one value of j, or $\perp$ if the entire output of the instruction does not depend on the user key.*

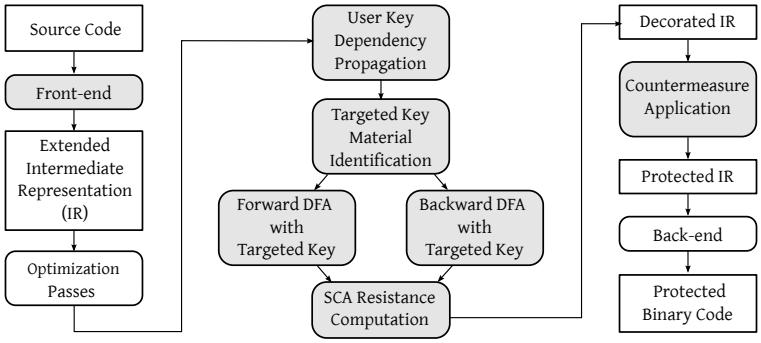


Figure 1: Representation of the LLVM compiler pipeline (added and modified passes in gray), including the passes implementing our DFA and code transformation to apply SCA countermeasures

We note that, given the SCA resistance of an instruction v, $\mathtt{r}(v)$, an adversary will require a computational effort $\Omega(2^{\mathtt{r}(v)})$ to perform a key retrieval guessing $\mathtt{r}(v)$ bits. The computation of the SCA resistance is performed through a sequence of static code analyses and transformations performed by the front-end and middle-end of the LLVM compiler infrastructure as depicted in Figure 1.

Compiler Front-End and Optimization Passes. First of all, we modified the `clang` LLVM front-end to recognize three attributes with the standard GNU syntax, which are employed by the programmer to mark i) the variables containing the block cipher input and ii) user key, and iii) the tabulated nonlinear functions named substitution boxes (SBoxes). To tag instructions in a way that survives the optimization passes of LLVM, we employ the input and key values as arguments of opaque intrinsics, defined to have memory affecting side effects. We run the entire set of optimizations of the middle end (-O3), bar the ones able to identify patterns in the code which can be substituted with a call to the C standard library (e.g., replacing a memory copy with a call to `memcpy`). The LLVM Intermediate Representation (IR) emitted by the front-end is in *Single Static Assignment* form: a 1:1 binding between an instruction $v \in V$ and its defined variable is present. We will employ the two concepts interchangeably from now on, and consider the IR as the program P out of which $\mathcal{G}_{\mathtt{C}}(V, A_{\mathtt{C}})$ and $\mathcal{G}_{\mathtt{D}}(V, A_{\mathtt{D}})$ are built.

User Key Dependency Propagation. The user key dependency propagation pass is the first one of our analysis. Its purpose is to compute the data dependency of the entire cipher and key schedule from the user (i.e., non expanded) key. To this end, we compute a partition of the nodes of $\mathcal{G}_{\mathtt{C}}(V, A_{\mathtt{C}})$ in three subsets, V_{ks}, V_c, V_a according to whether they belong to the key schedule, the cipher body, or to eventual ancillary code, respectively. Such a partition is computed as a simple forward DFA, of which the property is constituted by a two bit vector, the first one indicating if the node lies in V_{ks}, and the second one if it is in V_c. The function of the DFA equations sets the first bit only if all the nodes having their result used by the current instruction are in V_{ks}, while the second bit is set if at least one of the operands is in V_c. The initial state of the data-flow marks as belonging to V_c all the instructions tagged with our intrinsics, while the key schedule bit is set only on the key-load ones. The results of this analysis will be employed by the Forward/Backward DFA with Targeted Key passes (see Figure 1).

Subsequently, a second DFA computes the data dependency of each operation from the user key. To this end, the computed property is described as a $|v|\times|k|$ Boolean matrix $\mathbf{U}_v$. The DFA solver initializes, for all $v \in V$, the $\mathbf{U}_v$ matrices to zero, save for the ones bound to user key-load instructions, which have the appropriate $|v|\times|v|$ submatrix set to the identity matrix, to represent the 1:1 dependency of a user key bit from itself. The $f_v(\cdot)$ function of the forward DFA considers the user key dependencies

Algorithm 3.1: Target Key Material Identification

Input: $\mathcal{G}_{\mathtt{C}}(V, A_{\mathtt{C}})$: CFG of a block cipher with a $|k|$ bits user key; $\delta_s(v) : V \rightarrow \{0, \ldots, d_{max}\}$: map of a node $v \in V$ to its distance from v_{begin} of $\mathcal{G}_{\mathtt{D}}(V, A_{\mathtt{D}})$; d_{max} is the height of the DFG; $\mathbf{U}_v$: matrix of user key dependencies for $v \in V$
Output: T: set of elements of V_{ks} containing target key material, $\perp$ if no viable assignment
Data: $\mathbf{f}$, $\mathbf{o}$, $\mathbf{b}$: bit vectors of size $|k|$; $\mathbf{0}$, $\mathbf{1}$: constant $|k|$ bit vectors, zero- and one-filled.
F : Set of triples $(v, \mathbf{c}, \mathbf{l})$ where $v \in V$, where $\mathbf{c}, \mathbf{l}$ are Boolean vectors of length $|k|$ containing the subkey bits bound to v and left over as a result of an unconstrained choice

1 $\mathbf{b} \leftarrow \mathbf{0}$; $T \leftarrow \varnothing$; $F \leftarrow \varnothing$
2 **for** $d \leftarrow 1$ **to** d_{max} **do**
3 **foreach** $v \in V$ *s.t.* $\delta(v) = d$ **do**
4 $\mathbf{f} \leftarrow$ OrRowwise$(\mathbf{U}_v) \wedge \neg \mathbf{b}$
5 $\mathbf{o}, F \leftarrow$ GreedyOptimize$(v, \mathbf{U}_v, \mathbf{f}, F)$
6 **if** $\mathbf{o} \neq \mathbf{0}$ **then**
7 $\mathbf{b} \leftarrow \mathbf{b} \vee$ CheckConstraints$(v, \mathbf{f}, \mathbf{o}, F)$
8 $T \leftarrow T \cup \{v\}$
9 **if** $\mathbf{b} = \mathbf{1}$ **then return** T
10 **return** $\perp$

of the operands of v, and sets its $\mathbf{U}_v$ combining the ones of its operands. The user key dependencies of a single bit of the output of v, i.e., a row of $\mathbf{U}_v$, are obtained computing the elementwise Boolean-$\mathtt{or}$ combination of the rows corresponding to the bits of the operands required to compute it. If v is a bitwise operation, its $f_v(\cdot)$ function derives $\mathbf{U}_v$ matrix as the row-wise $\mathtt{or}$ of the matrices of its operands. By contrast, the $f_v(\cdot)$ for arithmetic operations take into account also the effects of borrows and carries. For an in-depth description of the individual $f_v(\cdot)$ functions we refer the reader to [1].

Target Key Material Identification. Once the first pass of the analysis has computed the contents of the $\mathbf{U}$ matrices for all the instructions, we proceed to the identification of the subkey bits which yield the best adversary choice during an attack. While such an identification is easy in the case of AES, as reported in [1], due to the nature of the cipher *keyschedule*, it is not always immediate to determine which set of intermediate variables are the most favorable to be guessed by the adversary to derive the user key when a generic cipher is considered. To compute this set, we rely on the conservative simplification that the keyschedule of the cipher is always invertible, assuming that the adversary knows an amount of key material which is both equal to the user key size and depending on all of it. Such a simplification favors the adversary, which, in case the aforementioned assumption does not hold, will be required to extract the entire key material produced by the keyschedule via SCA.

Consequentially, the set of key material bits which will be the target of the actual SCA needs to fulfill the following properties: *i)* its size is greater or equal than the user key size, *ii)* for each user key bit, at least one of its members depends on it, *iii)* the entirety of its members are not dominated by the remaining key material on the $\mathcal{G}_{\mathtt{D}}(V, A_{\mathtt{D}})$, *iv)* its size is minimized. Conditions *i)* and *ii)* ensure the computation of the user key employing the retrieved key material, while condition *iii)* ensures the minimization of the adversary effort. Condition *iv)* is imposed so that no key material which cannot be derived from SCA extracted one, is selected.
To derive the set of instructions $T \subset V_{ks}$ containing the target key material as output, the first step is to compute a map $\delta_s(\cdot) : V \rightarrow \mathbb{N}$ which expresses the distance of a given instruction from $v_{\mathtt{begin}}$ in the $\mathcal{G}_{\mathtt{D}}(V, A_{\mathtt{D}})$. Subsequently a map $\delta_e(\cdot) : V \rightarrow \mathbb{N}$ which expresses the distance from $v_{\mathtt{end}}$ is also computed. Such maps can be computed performing a breadth first visit of $\mathcal{G}_{\mathtt{D}}(V, A_{\mathtt{D}})$, following the definition-use chains in it. Two maps are required, as the optimal choice of an adversary for the target key material is different, depending on whether the data available to her is the input or the output of the cipher. For the sake of clarity, we consider from now on the former case, noting that the latter is analogous, provided that care is taken to reverse the direction of the analyses. Once the $\delta_s(\cdot)$ map is available, the target key identification pass proceeds to employ Algorithm 3.1 to compute the set T, initializing it as empty (line 1). To determine 1:1 binding between the user key bits and the target key material bits satisfying con-

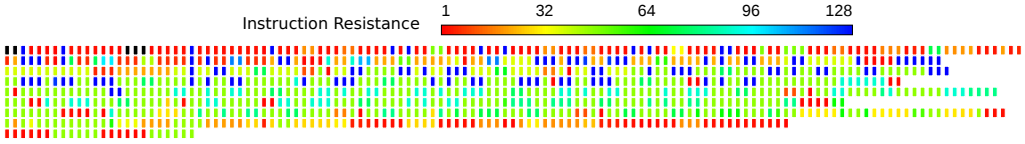
ditions *i)* and *ii)*, a $|k|$-bit sized vector $\mathbf{b}$ is employed to keep track of which user key bits have already been bound. Such a vector is initially set to $\mathbf{0}$ (line 1). The examination of the user key dependency matrix $\mathbf{U}_v$ of an instruction v may have a significant degree of freedom in choosing which user key bits should be bound to the target key material ones. Indeed, its $|v|$-bit output may depend on all the $|k|>|v|$ user key bits. Since a bijection is desired, Algorithm 3.1 will perform an arbitrary choice locally, and memorize it in the form of a triple $(v, \mathbf{c}, \mathbf{l})$ containing the instruction v, which user key bits were chosen to be bound $\mathbf{c}$, and which ones to be left $\mathbf{l}$, respectively. Such an information is stored in the set F, initially empty (line 1), which is exploited to optimize the target key material selection process.

After initializing $\mathbf{b}, T$ and F, Algorithm 3.1 examines the elements of V_{ks} starting from the ones closer to $v_{\texttt{begin}}$ in $\mathcal{G}_{\mathtt{D}}(V, A_{\mathtt{D}})$ (lines 2-10) and tries to bind all the user key bits as soon as possible. To this end, the computation starts with determining which of the user key bits, on which the output of v depends, are still free from binding (line 4). Such bits, which are deduced removing the bound ones from the row-wise Boolean-$\mathtt{or}$ reduction of $\mathbf{U}_v$, are stored in the $|k|$-bit long vector: $\mathbf{f}$.

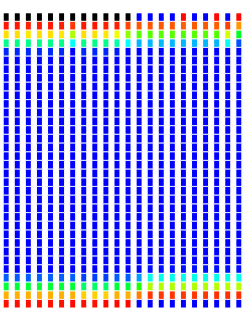
Subsequently, to the GREEDYOPTIMIZE procedure attempts to ameliorate the current user key dependency choice $\mathbf{U}_v$ for the current node v exploiting the information on which bindings have already been made, and which ones had freedom of choice in being done as memorized in $\mathbf{f}$ and F In particular, for each element $(v', \mathbf{c}, \mathbf{l})$ of F, GREEDYOPTIMIZE checks if *a)* some of the bits bound to v' can also be bound to the output of v, and *b)* some of the unbound bits in v' cannot be bound to v. If this is the case, it is possible to optimize the decision of the bits to be bound to the output of v having it relieve v' of some of its arbitrarily assigned bits, in exchange for v' taking care of bits which cannot be bound in v. Since the nodes of v are visited in increasing depth order, GREEDYOPTIMIZE effectively realizes swaps which guarantee condition *iii)*, i.e., no set of target key material bits dominates strictly the chosen one on the DFG. GREEDYOPTIMIZE returns both an optimized choice bit vector $\mathbf{o}$, and the updated set F. If the optimized binding selection contained in $\mathbf{o}$ suggests that at least a user key bit should be bound to the output of v, i.e., $\mathbf{o}\neq\mathbf{0}$, (line 6) Algorithm 3.1 proceeds to add the contents of $\mathbf{o}$ to the bound bits contained in $\mathbf{b}$ after verifying the feasibility of the binding invoking the CHECKCONSTRAINTS function. The CHECKCONSTRAINTS verifies that the amount of user key bits which should be bound in the outputs of v does not exceed $|v|$, and, if that is the case, limits the amount of the bound bits to $|v|$, adding an element to the F set to memorize the choice made. Upon completing the visit of all the nodes with a given depth, Algorithm 3.1 checks whether all the user key bits are bound (line 9) and, if that is the case, returns the set T. The reason for checking this condition at the end of the visit of the set of nodes having the same depth is to allow the possible optimization of all last level assignment to take place.

Forward/Backward DFA with Targeted Key and SCA Resistance Computation. After computing the set of target key material, the analysis passes proceed with the computation of the data dependencies of all the nodes in V_c from the selected target subkey bits. Such a computation is performed employing the same DFA used to determine the dependencies from the user key, taking care of considering the target key material data dependencies as the ones to be propagated. The analysis is performed both in the form of a forward DFA, and a backward one considering the data dependencies of the inverse operations, starting from the cipher output. The final result of the computation are two Boolean matrices, $\mathbf{F}_v$ and $\mathbf{B}_v$, containing the data dependencies of each instruction from the target subkey bits. To the end of computing the SCA resistance of an instruction v, the analysis computes the minimum, non null, number of key bits which depend on a single output bit of v across both matrices. We note that having an instruction output bit which depends on no key bits results in the value being irrelevant for SCAs, as it does not contain any key.

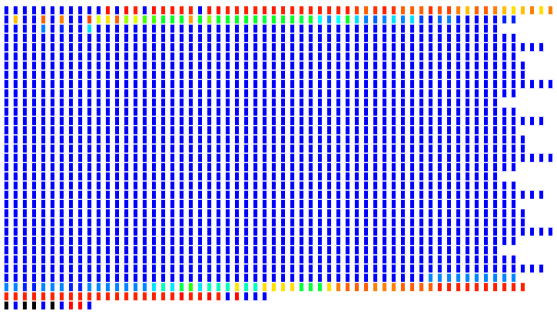
Countermeasure Application. Finally, after the computation of the per-instruction SCA security margin, we realized a code transformation pass applying systematically the ISW masking scheme [11] to all the instructions having a SCA resistance strictly lower than the key size of the cipher. The same pass also employs the countermeasure proposed by Coron [9] to secure load operations from SBoxes, exploiting



(a) MISTY1(8 rounds)



(b) Simon (68 rounds)



(c) Serpent (32 rounds)

Figure 2: Representation of the SCA resistance: each tile represents an instruction, with the color encoding the resistance value from 1(red) to 128(blue). Black tiles are key-independent instructions

the intrinsic markers to locate them. Share splitting and recombination instructions are automatically inserted at the borders of the region of the program to be protected.

4 Experimental Evaluation

In our case study, we chose a general purpose block cipher, Serpent, and two lightweight ones, MISTY1 and Simon, implemented in C language without any architecture-specific optimization.

We chose Serpent due to its significant security margin, which resulted in the design being selected as a finalist in the AES contest [5]. Serpent is a substitution-permutation network, with a 128-, 192- or 256-bit key, operating on a 128-bit block in 32 rounds. We will consider the 128-bit key for the sake of comparison. Serpent is endowed with a substitution stage which allows a *bitsliced* implementation (no SBoxes), while its diffusion layer achieves complete diffusion in a single round.

MISTY1 is a lightweight cipher designed in 1995, and described in the IETF RFC 2994 [13]. It has a Feistel network structure with a 128-bit key, a 64-bit block and an advised number of rounds equal to 8. Whilst being subject to some cryptanalytic attacks, MISTY1 forms the mainstay of KASUMI, later standardized as A5/3, the cipher used to encrypt third generation mobile networks.

Simon is a lightweight block cipher released in 2013 by Beaulieu et al. from the U.S.A. National Security Agency [7] and intended to provide sound security in constrained environments. It is a Feistel network with a very simple `F`-function, made of a single bitwise `and`, an exclusive-`or`, and three bitwise rotations, plus a key addition. It may act on a variable block size, and with a variable key size: we consider the 128-bit block, 128-bit key, 68 rounds variant.

Figure 2 reports the graphical representation of the results of our security oriented data-flow analysis on Serpent, MISTY1 and Simon. The ciphers are represented as a sequences of colored tiles, one per instruction, organized in program order from left to right, and from top to bottom. The figure shows the SCA resistance of each instruction executed during a cipher run, with colors ranging over a chromatic

Table 1: Performance results for the considered block ciphers on the Cortex-M4 and MIPS32 platforms

ISA	Masking order & region	Execution time (ms) AES [1]	Serpent [This Work]	Simon [This Work]	MISTY1 [This Work]	Code size (kiB) AES [1]	Serpent [This Work]	Simon [This Work]	MISTY1 [This Work]
ARMv7 Cortex-M4	None	0.076	0.089	0.071	0.028	7	9	8	2
	1-tailor	39.3	0.30	0.41	27.7	28	13	14	13
	1-full	103.7	1.90	1.45	65.9	57	40	31	13
	2-tailor	118.7	0.56	0.90	80.79	67	18	22	28
	2-full	299.6	4.36	3.49	194.1	144	79	67	27
	3-tailor	250.5	0.89	1.54	169.6	118	24	32	46
	3-full	621.2	7.42	6.20	411.0	267	127	109	44
MIPS	None	0.009	0.008	0.008	0.002	14	13	13	4
	1-tailor	25.92	0.75	1.09	23.96	48	24	23	23
	1-full	64.40	5.62	4.46	66.36	92	65	55	23
	2-tailor	99.51	1.74	2.74	91.32	92	32	39	46
	2-full	246.38	13.19	11.24	259.51	198	128	125	46
	3-tailor	221.54	2.97	4.85	202.23	161	43	57	79
	3-full	547.13	22.58	20.25	581.29	352	214	195	77

scale from 1 (red) to 128 (blue). The higher the resistance value, the higher the computational effort required to perform a secret key retrieval. The maximum resistance value (128 in our case) corresponds to the computational effort required for an exhaustive user key search [6]. In Figure 2, a row represents one *cipher round* for both Serpent and MISTY1, while it represents two for Simon. From Figure 2 (a), it appears that the internal structure of MISTY1 is unfriendly towards tailored protection against SCA. Indeed, if a security margin against SCAs matching the one against theoretical attacks is desired, the vast majority of cipher instructions requires the application of countermeasures. By contrast, both Simon and Serpent exhibit a very fast combination of the entire key material with the cipher state, leading to the need of protecting only 16 rounds and 4 rounds out of 68 and 32, respectively. Such a gap in requirements is due to the sound diffusion strategy adopted by both ciphers, which turns out to be more effective in the case of Serpent thanks to its well structured linear diffusion layer. The slower diffusion in Simon is to be expected as the only operations performing it are the bitwise rotations of its `F`-function.

Performance Evaluation. We conducted our performance evaluation measuring both implementations of the aforementioned three ciphers plus the one of AES-128 cipher. To provide a fair comparison with the results of [1], which used a different protection scheme for the SBoxes, we collected the results for AES employing the same protection scheme employed for the other ciphers, i.e., [9] (see Section 3). Our enhanced analysis identified the target key material in AES-128 matching the selection in [1].

The first evaluation platform is the STM32F4Discovery board equipped with a STM32F407 μC based on a 120MHz ARMv7 Cortex-M4, and endowed with 128kiB of SRAM, 1MiB of Flash and a hardware Random Number Generator (RNG). The binaries are run on the bare metal, and the timings are gathered measuring the time between the assertion and deassertion of a GPIO on 30 runs of the cipher. The second platform is an Imagination Creator CI20 board equipped with an Ingenic JZ4780 SoC. The SoC is based on a dual core MIPS32 clocked at 1.2GHz, with a 32k L1 I- and D-cache, a 512k L2 I&D cache, and 1GiB of DDR3 RAM, running Debian wheezy. Timing measurements where collected using the Linux kernel timer `CLOCK_PROCESS_CPUTIME_ID`, which only accounts for cycles attributed to the current process. To compensate for the higher variability of the environment, timings have been averaged over 10^4 measurements. To supply random values for the masking countermeasures, we employed the hardware RNG on the Cortex-M4, and the system-wide RNG on the MIPS based one.

Table 1 reports the results of our experimental campaign. Applying tailored SCA countermeasures, i.e., applying them to all instructions with a SCA resistance lower than the key size, allows to obtain significant performance improvements over a fully protected implementation. Block ciphers which em-

ploy lookup tables, i.e., AES and MISTY1 have the largest absolute gain in execution time, although the smallest relative one. Moreover, block ciphers which do not use lookup tables for non-linear functions are significantly faster when endowed with SCA countermeasures. It is worth noting that both Serpent and Simon, which are slower than MISTY1 in their unprotected implementation, are at least one order of magnitude faster when protected. Indeed, it is interesting how Serpent outperforms Simon on both platforms if tailored SCA countermeasures are applied, despite the latter being explicitly designed as a lightweight cipher, and the former not being so. Finally, it is interesting to note how the $10\times$ difference in clock speed between the two platforms yields only moderate speedups for the fully protected variants of AES and MISTY1. This behavior is caused by the increase in the size of the lookup tables required by the share splitting of [8], which adversely effects caches. Table 1 reports also the figures obtained measuring the size of the `.text` segment of the emitted binaries for both platforms. Such a metric, which is typically critical in code-memory constrained devices such as μCs is still relevant on high-end embedded platforms, as a small code segment fits more easily in the L1 instruction cache. Considering this metric, MISTY1 exhibits the interesting feature of having substantially the same code size, regardless of the fact that a tailored or a complete application of SCA countermeasures is made. This is due to the cost of the share splitting and recombination instructions which are needed when a transition from an protected instruction to an unprotected one is made. However, despite retaining the smallest code size among fully protected implementations, MISTY1 is bested by both Simon and Serpent if a tailored protection is applied. In particular, a third-order tailored masking applied to Serpent results in a code size smaller than the one of a first-order protected AES on both platforms.

5 Concluding Remarks

In this work we presented an enhancement to the existing security oriented data-flow technique, which allows the automatic identification of the target key material in a block cipher. Such an enhanced analysis allowed us to analyze block ciphers with nontrivial target key material choices, and to evaluate a tailored protection to their implementation as a secure alternative to a code morphing approach such as [4]. The relevance of the extended analysis is witnessed by the experimental results showing how a high-security, general purpose cipher, Serpent, which is outperformed by a factor of four by a lightweight one, MISTY1, achieves speedups up in the $93\times$–$190\times$ range when protected implementations are considered, without losses on the overall computational security margin [6].

References

[1] Giovanni Agosta, Alessandro Barenghi, Massimo Maggi, and Gerardo Pelosi. Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 81:1–81:6, 2013.

[2] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. A code morphing methodology to automate power analysis countermeasures. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 77–82. ACM, 2012.

[3] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. A Multiple Equivalent Execution Trace Approach to Secure Cryptographic Embedded Software. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 210:1–210:6. ACM, 2014.

[4] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. The MEET approach: Securing cryptographic embedded software against side channel attacks. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(8):1320–1333, 2015.

[5] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the advanced encryption standard. *NIST AES Proposal*, 174, 1998.

[6] Alessandro Barenghi and Gerardo Pelosi. On the Security of Partially Masked Software Implementations. In Pierangela Samarati, editor, *SECRYPT 2014 - Proceedings of the 11th International Conference on Security and Cryptography, Vienna, Austria, 28-30 August, 2014*, pages 138:1–138:8. SciTePress, 2014.

[7] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013.

[8] Jean-Sébastien Coron. Higher Order Masking of Look-Up Tables. In *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 441–458. Springer, 2014.

[9] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In *CHES 2014*, pages 188–205. Springer, 2014.

[10] Flavio D. Garcia, David Oswald, Timo Kasper, and Pierre Pavlidès. Lock it and still lose it - on the (in)security of automotive remote keyless entry systems. In *USENIX Security 16, USA, Aug.*, 2016.

[11] Yuval Ishai, Amit Sahai, and David Wagner. Private Circuits: Securing Hardware against Probing Attacks. In D. Boneh, editor, *CRYPTO*, volume 2729 of *LNCS*, pages 463–481. Springer, 2003.

[12] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, 2007.

[13] Mitsuru Matsui and Hidenori Ohta. A description of the MISTY1 encryption algorithm, 2000.

[14] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Advances in Cryptology–CRYPTO 2015*, pages 764–783. Springer, 2015.

[15] Daehyun Strobel, David Oswald, Bastian Richter, Falk Schellenberg, and Christof Paar. Microcontrollers as (in)security devices for pervasive computing applications. *Proceedings of the IEEE*, 102(8):1157–1173, 2014.