Encasing Block Ciphers to Foil Key Recovery Attempts via Side Channel*

Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale
Dipartimento di Elettronica, Informazione e Bioingegneria – DEIB, Politecnico di Milano
email: name.surname@polimi.it

Abstract Providing efficient protection against energy consumption based side channel attacks (SCAs) for block ciphers is a relevant topic for the research community, as current overheads are in the $100\times$ range. Unprofiled SCAs exploit information leakage from the outmost rounds of a cipher; we propose a solution encasing it between keyed transformations amenable to an efficient SCA protection. Our solution can be employed as a drop in replacement for an unprotected implementation, or be retrofit to an existing one, while retaining communication capabilities with legacy insecure endpoints. Experiments on a Cortex-M4 μC, show performance improvements in the range of $60\times$, compared with available solutions.

1 Introduction

Embedded computation devices in the modern age are becoming increasingly pervasive and are often required to perform security critical tasks where providing data or communication confidentiality is a mandatory requirement. The prime choice to do so is to employ symmetric block ciphers, due to their high efficiency even when limited computing resources are available, e.g., on low end microcontrollers and RFID chips. As a direct consequence of the pervasiveness and low cost of the embedded devices, a practical attacker model is the one considering the possibility of seizing one of such devices with the purpose of extracting the secret key of the block cipher, thus invalidating completely the provided security guarantees. Direct access to the device allows the attacker to exploit information encoded in environmental parameters of the computation, e.g., its energy consumption. Such an attack is commonly known as a passive Side Channel Attack (SCA) and has been shown to be remarkably efficient in breaching the security of widely deployed products [15]. SCAs exploit the information measured on a computing device to retrieve the secret key guessing it piecewise. In particular, the dependence of the energy consumption of a device from the values of the data being

*This is a preprint version of the paper published in the proceedings of the ICCAD conference. The final version can be found at: http://dx.doi.org/10.1145/2966986.2967033

processed is employed as an unintended communication channel. To this end, an intermediate value of the computation, which can be obtained as a combination of the input and a small amount of the secret key (e.g., a single-byte `xor`), is guessed for all the possible values of the involved key portion. Such guesses are then employed to predict the side channel measurement, and compared against the actual one via a statistical test, obtaining a ranking of the tested key guesses: the one providing the best fitting prediction is the one relying on the actual key. The prediction of the side channel measurement is typically done with an *a-priori* model, e.g., the Hamming weight of the computed value or its Hamming distance from the value previously stored in the same register. The alternate way, involves exploiting an *a-posteriori* model obtained profiling the side channel behavior of another instance of the device under attack, which is fully controlled by the attacker [15].

Provably secure countermeasures against such attacks [12,15,19,20] provide a way of performing the computation of the block cipher in a redundant fashion, employing the addition of unpredictable random values to it. The modified computation strategy provides a correct output, while ensuring that intermediate values depend on both the inputs and the random values added.

Consequentially, to lead a successful SCA when these countermeasures are in place, the information coming from the side channel leakage of more than one intermediate value must be combined to build a model which is independent from the random values involved. Such a combination is performed subtracting the measurements (e.g., power consumption) obtained from the side channel when the materialization of the random values is performed, from the measurement where they have been all combined to the intermediate value to be attacked. The result is shown to be proportional to the `xor` combination of all the aforementioned values [15], which does not contain the contribution of the random values, as they are added twice to the same intermediate value (once by the protected algorithm and once by the combination). The difficulty of such an attack, known as *High-Order* (HO) attack, is quantified in terms of either the number of different side channel leakage values, d, which are combined to perform it, or of the degree, d, of the statistical moments of the leakage at a single point in time [15].

In [17], the authors show that, in case it is possible to extract the Hamming weight of the values being loaded and stored into a μC SRAM exploiting information coming from a profiled attack, a SCA will succeed in recovering the secret key employed in the device even in the presence of arbitrarily complex countermeasures, and without knowledge of the inputs and outputs. However, the profiled attack in [17], which succeeded on an 8-bit PIC16F877 μC (etched with a 500nm technology node), has been proven no longer feasible on a more recent 32-bit ARMv7 CPU [4], as a consequence of the difficulty of obtaining accurate enough profiling information induced by both technology scaling and a higher complexity of the target device. The authors of [18] report that in a $1,530$ transistors, 65nm, ASIC implementation of the AES cipher, process variation prevented the extraction of the correct Hamming weight profile. In the light of these results, we will be considering the widely accepted attacker model where only the knowledge of either the input or the output values of the cipher is required, and the computational effort required to perform an (un-profiled) SCA against an unprotected implementation grows exponentially in the number of key bits to be guessed. In this model, it is possible to reduce the application of the countermeasures only to the

first and last rounds of a block cipher, as predicting a value in the remaining portion of the cipher would imply guessing the value of the entire key [1, 2]. The computational burden required by the countermeasures grows with the maximum order of the attack d against which the designer chooses to be protected. Indeed, the number of measurements (which depends on the noise standard deviation) required to perform a d-th order attack against an implementation grows exponentially with the exponent being $d+1$ [16].

Despite the possibility of reducing the application of provably secure SCA countermeasures only at-the-ends of the block cipher, execution time slowdowns greater than $100\times$ are not uncommon in software implementations, due to the difficulty of protecting either the nonlinear operations or the table lookups present in the cipher [1, 7]. In the case of hardware implementations such computation time penalties are less severe, although this is usually obtained at the cost of a significant increase in the occupied area.

A promising research direction is the one aiming at designing a totally novel cipher, which is conceived to be both mathematically secure and easy to protect against SCA. A prominent example of such ciphers is the one proposed in [10]. However, such an approach needs to face the tradeoff between the mathematical strength of the cipher (e.g., against linear and differential cryptanalyses) and the efficiency of the SCA protection (provided by a low algebraic degree of the nonlinear functions employed in the cipher design) [11].

Our proposal in this work follows a different direction, namely, we encase a block cipher within two *keyed transformations*, which are easy to protect against side-channels and do not affect adversely the mathematical security of the encased primitive. Protecting the keyed transformations with a provably secure SCA countermeasure will thus leave an attacker unable to break it, and thus with the only option of guessing the entire key employed in one of them to lead an SCA against the encased cipher. This approach has the advantage of exploiting existing and well scrutinized block ciphers to provide the mathematical security required from the construction, although it results in the output value of the augmented primitive not matching the one of the encased cipher alone. From a more practical perspective, it is possible to employ the proposed augmented cipher as a drop-in replacement for an unprotected one, as it matches its block size, while the extra key material can be securely obtained employing the approach described in [14]. In particular, the keys of both the cipher and the keyed transformations can be computed from the original secret key feeding its concatenation to two, different and equally long prefixes to two instances of a cryptographic hash function. The proposed approach can be implemented in both hardware and software solutions, taking care of implementing the encasing primitive accordingly. A viable software implementation is also to retrofit existing block cipher instances with the proposed protection, both in case of software libraries and separate block cipher co-processors, as it may be realized wrapping at-the-ends the existing artifact computing the primitive. Consequentially, it is possible to have an implementation which is protected, and may function (if needed) in a legacy-supporting (i.e., unprotected) fashion, skipping the computation of the keyed transformations. This strategy is akin to the one of the triple DES (TDEA [13]) cipher in encryption-decryption-encryption mode, which was proposed both to strengthen the aging single DES cipher and to provide a fallback compatibility mode employing the same key in all three DES executions. We note that, in

a similar fashion to triple DES, our encased block cipher proposal should be treated as an atomic primitive when employed in any mode of operation to process inputs larger than a single block.

Contributions. In this work we propose to provide protection against energy-consumption-based SCAs encasing a cipher implementation between two keyed transformations, and applying countermeasures only to them. We designed an efficient-to-protect keyed transformation which allows us to achieve performance gains greater than $60\times$, when compared with the current state of the art of block cipher implementations protected with provably secure countermeasures, on our experimental platform. We also propose an automated method to remove *transition leakage* from software implementations through zeroing out the contents of a the destination register of an operation, allowing the developer to consider *value leakage* alone when protecting the keyed transformation. We provide an implementation of our contributions as a `C++` template library and a modified LLVM compiler toolchain, and evaluate its effectiveness with respect to the current state of the art of the side channel countermeasure approaches.

Organization of the work. The remainder of the work is organized as follows: Section 2 provides a recap of the state of the art of provably secure countermeasure strategies against side channel, the concept of transition-leakage vulnerability and our proposed solution to cope with it. Section 3 describes our proposed encased cipher construction, detailing the design of the encasing keyed transformation and its security guarantees, together with the methodology applied to automatically prevent transition leakage due to register reuse in software implementations. Section 4 provides an experimental campaign on a commercial grade platform, comparing our performance results with the current state of the art proposals. Finally, Section 5 draws our conclusions.

2 SCA Countermeasures

In this section we will provide the background on the SCA countermeasures, which we will be employing to secure the keyed transformation, and their computational cost. Among the significant number of countermeasure strategies proposed in open literature, we chose the two approaches providing a constructive scheme to design a d-th order resistant countermeasure. We also describe our contribution on effectively and efficiently implementing them in software, preventing the so-called *transition leakage*.

Provably Secure Countermeasures. Among the possible protection schemes, the *Ishai-Sahai-Wagner masking* (ISW masking) [12] and *Threshold Implementations* (TI) [19] are the two methodologies providing constructions backed by a security proof stating their soundness up to d-th order attacks. Their structure is related, and the explicit link between them has been analyzed in [19], highlighting possible implementation issues in hardware and software.

Both schemes rely on encoding each one of the inputs to the algorithm to be protected as a set of $s \geq 2$ shares taking uniformly distributed bit values during the computation. Typically, such encoding is performed adding via bitwise `xor` $s-1$ independent random values to each input to obtain the first share, and taking the $s-1$ random values as the remaining ones. Once the encoding is completed for all inputs, the computation of the algorithm itself is adapted to be performed on the encoded values, obtaining a result which is also split over multiple shares. The adaptation of the algorithm is

performed considering it as a set of Boolean functions of its inputs. The result is reconstructed adding via bitwise `xor` all its shares. Care should be taken not to combine all the shares of an intermediate value of the algorithm, lest its value be disclosed.

The ISW masking [12] provides a constructive method to compute Boolean `and` and `not` bitwise operations on share-split variables given the desired protection order d. Its security proof relies on the fact that the values of any d variables of the adapted algorithm computation (in any given time instant) could be the simulated by the output of a random number generator (RNG). As a consequence, a d-th order attacker has no knowledge whatsoever on the actual values being computed by the algorithm. To achieve the aforementioned security level, the ISW masking requires to split each of the unencoded inputs into $s{=}2d{+}1$ shares. The authors of [12] also propose a tweaked masking scheme (tweaked ISW masking henceforth) allowing to limit the number of shares to $s{=}d{+}1$, at the cost of a higher pressure on the RNG.

TIs [19] take a different approach at providing the desired d-th order security, modifying the algorithm computation so that, in no time instant the computation of a share of an intermediate result depends on more than $s{-}d$ shares of the encoded input, a property named *d-non-completeness*. Applying the scheme to turn the algorithm into a TI yields a computation of the shares of the result, both respecting the d-non-completeness property, and satisfying the fact that the `xor` recombination of all of them produces the correct value, i.e., the TI is *correct*. It is known that, for a generic Boolean function of degree t, it always exist a d-secure TI which encodes each input into $s_{\texttt{in}}{=}td{+}1$ shares, yielding the result split over $s_{\texttt{out}}{=}\binom{td+1}{t}$ ones. However, there is no known result on the optimal values of $s_{\texttt{in}}$ and $s_{\texttt{out}}$ to minimize $s_{\texttt{in}}{+}s_{\texttt{out}}$, for a TI of a Boolean function [5], other than having $s_{\texttt{in}}{\geq}d{+}1$ and $s_{\texttt{out}}{\geq}d{+}1$ to enable d-non-completeness. The authors of [19] provide a specific TI to compute the Boolean `and` (i.e., a Boolean function with degree $t{=}2$) with $d{=}1$ and $d{=}2$, employing $s_{\texttt{out}}{=}s_{\texttt{in}}{=}3$ and $s_{\texttt{out}}{=}s_{\texttt{in}}{=}5$, respectively. The TI of the Boolean `not` operation can be trivially obtained complementing the value in a single share of a split variable.

Both in the case of the ISW masking and the TI, computing a Boolean `and` on share-split variables requires $O(s_{\texttt{in}}^2)$ operations, computing a `not` requires $O(1)$ operations, and computing an `xor` requires $O(s_{\texttt{in}})$ operations. The reason for the lower cost of the `xor` computation is the fact that the share split encoding of the input itself is performed via `xor`. Consequentially, a Boolean function having a low degree when expressed in Algebraic Normal Form (ANF) can be protected more efficiently than another one having a higher degree. We considered both ISW masking and TIs as protection strategies for our keyed transformation, as the tweaked ISW masking provides a scheme to perform a d protected computation with only $s{=}d{+}1$ shares, while no TI has been proposed to do so for all values of d. By contrast TIs are computationally cheaper w.r.t. an ISW implementation with the same number of shares by a factor linear in $s_{\texttt{in}}$ depending on the specific form of the TI, possibly allowing a better computational tradeoff at low ds. To provide a fair evaluation of their efficiency, in Section 4 we compared both approaches for d between 1 and 4.

Whenever it is needed to protect a tuple of Boolean functions acting on the same inputs, as in the case of the nonlinear layer of a block cipher (e.g., the AES SUBBYTES), it is possible to represent the functions as a lookup table, and protect the `load` operations performed from it. In [7] a protection scheme for such `load` operations is proposed, and proven secure under the same attacker model as the ISW masking, showing

that such a protection technique is profitable whenever computing the tabulated outputs of the Boolean function requires a significant amount of `and` operations. In particular, the cost of a protected `load` operation with the algorithm in [7] is proportional to $O(4s_{\mathtt{in}}^2 w)$, where w is the number of elements of the lookup table.

Transition Leakage Vulnerability. Both ISW masking and TI tackle the issue of making the information coming from the intermediate values of a computation, also known as *value leakage*, useless for an attacker. While this provides security in fully combinatorial hardware, in case memory elements are reused in a sequential implementation, a new information is leaked on the side channel, namely the so-called *transition leakage*. Such a leakage is typical of software implementations, where it occurs often due to the natural reuse of the registers, which happens in a general purpose CPUs, when the liveness interval of the variable allocated into one of them ends. Whenever this happens, the general purpose register will be reused to store the value pertaining to a different variable, resulting in a side channel leakage proportional to the Hamming distance between the two values. Although the frequency of this reuse highly depends on the amount of register pressure of the code point, and the register allocation policies of the compiler, it is possible that two shares of the same variable are stored in the same register thus yielding a leakage of their `xor` combination, effectively undermining both the ISW masking and TI countermeasures.

A quantification of the reduction of the security margin due to transition leakage is provided in [3], where the authors prove that a countermeasure scheme with a d-th order security margin against value leakage will provide $\lfloor \frac{d}{2} \rfloor$-th order security against transition leakage. In this work we provide a contribution to mitigate this security margin loss for software implementations, providing a solution which is more efficient than doubling the order of the protection of the countermeasure. This solution relies on the following:

Lemma 1 (From transition- to value-leakage)**.**
Let $\mathcal{V}$ be the set of intermediate variables of a d-th order protected implementation and $\mathcal{T}$ the set of expressions obtained combining via `xor` every variable pair $(v_1, v_2) \in \mathcal{V} \times \mathcal{V}$. The transition leakage, i.e., the one stemming from elements of $\mathcal{T}$, is a subset of the value leakage, i.e., the one stemming from elements $\mathcal{V}$, if $\forall\, t \in \mathcal{T}$ at least one of the two variables in t is substituted with zero.

Proof. Let $t \in \mathcal{T}$ be an expression $t = v_a \oplus v_b$, with $v_a, v_b \in \mathcal{V}$. Replacing v_b with the constant 0, the expression t yields $t = v_a \oplus 0 = v_a$ for any possible value of v_a, and is thus contained in $\mathcal{V}$. Analogously, replacing v_a yields $t = v_b \in \mathcal{V}$. Each element $t \in \mathcal{T}$ is contained in $\mathcal{V}$ after the substitution is performed on either one of the variables present in t. □

A practical, and conservative, way to meet the condition specified in Lemma 1, is to materialize the constant 0 in the destination register `rd` of any CPU instruction before a new value is stored into it. This can be obtained efficiently either storing in `rd` the value `rd xor rd`, or copying the value 0 from a fixed value register, whenever this is available (e.g., the register `R0` in the MIPS ISA), and ensuring that the register allocator never reuses one of the operand registers as the destination. We note that `load` and `store` operations can be protected from transition-based leakage taking care of loading a 0 value stored in memory, and storing a 0 in the destination location, respectively. Such an action will reset the value of the memory data register on the CPU side, before

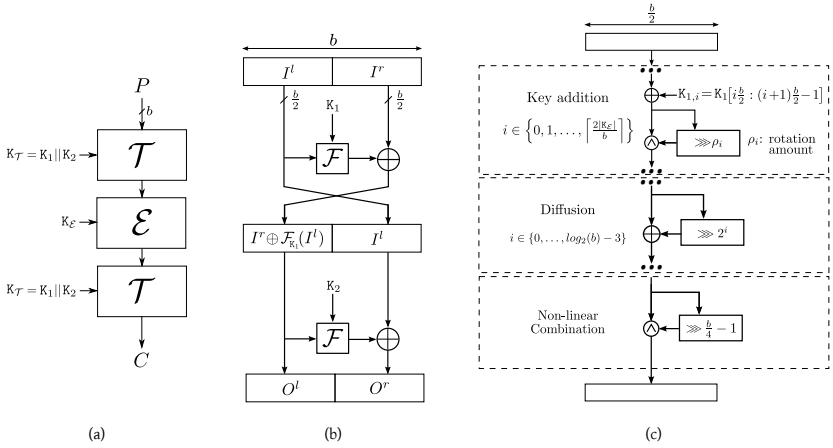

Figure 1: Proposed encasing for a block cipher $\mathcal{E}$, with key $\mathtt{K}_{\mathcal{E}}$ and block size b, within two keyed transformations $\mathcal{T}$, with key $\mathtt{K}_{\mathcal{T}}$, $|\mathtt{K}_{\mathcal{T}}| \geq |\mathtt{K}_{\mathcal{E}}| \geq b$ (a), detail of the keyed transformation $\mathcal{T}$ (b), and detail of the Feistel $\mathcal{F}$ function in $\mathcal{T}$, taking $I^l, \mathtt{K}_1$ as input (c)

a further `load`/`store` operation is attempted. We also note that, in case of `load` operations, the 0 value should be loaded from the same memory bank of the `load` to be protected, so that, the memory data register present on the memory bank is also set to the constant value. Augmenting the protected code with such instructions can be effectively, and automatically, done as a compiler pass. Such a program transformation pursues the direction of providing reduced overhead and low development time SCA countermeasure application pointed out in several works, such as [3]. The observation that pre-charging a register prevents transition leakage has been proven effective in [6]. In particular, in [6] the so-called *random pre-charging* strategy is applied to a fully unprotected implementation, and realized filling the contents of a register with random values obtained from an RNG. Employing random pre-charging in our case would still be effective, although at the unneeded cost of a significant amount of randomness from the RNG.

3 Augmented Cipher Construction

In this section we provide the details of the encasing strategy proposed to protect a block cipher $\mathcal{E}$, describing the keyed transformation $\mathcal{T}$ employed to do so, detailing its functional and security features, and the added SCA resistance. We report the methodology to provide automated support for transition leakage mitigation modifying the compiler employed to produce the binaries.

3.1 Encasing Strategy

Willing to protect a block cipher $\mathcal{E}$ with a b-bit wide block and secret key $\mathtt{K}_{\mathcal{E}}$, we propose to encase it between two instances of a b-bit wide keyed transformation $\mathcal{T}$ employing a key $\mathtt{K}_{\mathcal{T}}$ (with $|\mathtt{K}_{\mathcal{T}}| \geq |\mathtt{K}_{\mathcal{E}}|$), so that the plaintext P fed to the construction is

first processed by $\mathcal{T}$, and its output is fed into the encased cipher $\mathcal{E}$. Subsequently, the output of $\mathcal{E}$ is employed as input to another instance of $\mathcal{T}$ to produce the ciphertext C, as depicted in Figure 1(a).

In [9], the authors show that a construction made of a single, fixed, bijective function encased between two `xor` KEY ADDITIONs is necessary to properly define a block cipher primitive, as removing each one of them results in a cryptographically broken cipher. A somewhat similar intuition was exploited in [14], where an efficient method to strengthen the DES cipher against bruteforce attacks was proposed. The authors of [14] propose to encase the DES cipher between two KEY ADDITIONs (via `xor`) involving 64 extra key bits each, thus raising the security margin against exhaustive key search to $\approx 2^{120}$ trials from the original 2^{56} ones.

The design presented in this paper differentiates from both [9] and [14] in assuming that the primitive to be protected $\mathcal{E}$ is secure, and relies on it to provide the mathematical security of the construction. Indeed, our approach aims at endowing the encased primitive with side channel resistance, a feature that no current block cipher is enjoying by construction.

3.2 Keyed Transformation

To the end of providing side channel resistance to the block cipher primitive of choice $\mathcal{E}$, encasing it between a simple pair of bitwise `xor` KEY ADDITIONs as proposed in [14], and endowing such KEY ADDITIONs with provable side channel protection in their implementation, would not reach the purpose. Indeed, such a strategy would only increase by one the number of key bits to be guessed in an SCA aiming at the ends of the encased cipher [1]. In designing a more complex encasing transformation $\mathcal{T}$, it should be kept into consideration that the side channel protections applied to it have a cost growing quadratically in the degree of the ANF representation of its output bits (see Section 2). To this end, reducing the number of Boolean `and`s in its ANF expression will provide substantial performance benefits.

Our design of the encasing keyed transformation $\mathcal{T}$ provides the following functional and security features:

(F1). $\mathcal{T}$ is bijective for any given $\mathtt{K}_{\mathcal{T}}$.

(S1). The cascade $\mathcal{T}\mathcal{E}\mathcal{T}$ presents a security margin against exhaustive key search greater or equal than the one of $\mathcal{E}$ alone.

(S2). $\mathtt{K}_{\mathcal{T}}$ cannot be derived from P and C without an exhaustive performing an key search of $\mathtt{K}_{\mathcal{E}}$, assuming no side channel information is used.

(S3). An SCA predicting a single input/output bit of $\mathcal{T}$ should require guessing at least $|\mathtt{K}_{\mathcal{E}}|$ key bits of $\mathtt{K}_{\mathcal{T}}$.

(S4). An SCA exploiting an `xor`-combination of the side channel information coming from a set of output bits of $\mathcal{T}$ should require guessing at least $|\mathtt{K}_{\mathcal{E}}|$ key bits of $\mathtt{K}_{\mathcal{T}}$.

The resistance of $\mathcal{T}$ against statistical mathematical cryptanalysis techniques (e.g., against linear, differential and impossible cryptanalysis) is not required as such techniques need pairs of inputs-outputs coming from $\mathcal{T}$, to which the attacker has no access to in our scenario.

Figure 1(b) and Figure 1(c), report the detailed structure of the keyed transformation $\mathcal{T}$, of which we will now describe how it has been designed to have the aforementioned features, and which aspects of the security guarantees are obtained through them.

The keyed transformation $\mathcal{T}$ is a 2-round Feistel network, employing a $2|\mathtt{K}_{\mathcal{E}}|+b$-bit long key $\mathtt{K}_{\mathcal{T}}$ split into two halves $\mathtt{K}_1$, $\mathtt{K}_2$ each one used in a separate Feistel function instance $\mathcal{F}$ (see Figure 1(b)). The choice of a Feistel network design for $\mathcal{T}$ was made to allow freedom in the design of the Feistel $\mathcal{F}$, as the network structure guarantees feature **(F1)** for any possible choice of $\mathcal{F}$. In particular, $\mathcal{T}^{-1}$ is obtained employing the same structure of $\mathcal{T}$, with the keys $\mathtt{K}_1$,$\mathtt{K}_2$ input in reverse order. As a consequence, it is always possible to decipher a plaintext encrypted with the keyed transformation $\mathcal{TET}$ via the inverse transformation $(\mathcal{TET})^{-1}=\mathcal{T}^{-1}\mathcal{E}^{-1}\mathcal{T}^{-1}$, employing $\mathtt{K}_{\mathcal{E}}$ and $\mathtt{K}_{\mathcal{T}}$ properly. Note that alternate constructions for $\mathcal{T}$, e.g., by employing a Substitution-Permutation Network, may exist: the investigation of a general form of $\mathcal{T}$ is beyond the scope of this work.

Concerning feature **(S1)**, it is crucial for the mathematical security of the $\mathcal{TET}$ construction that the application of the keyed transformation $\mathcal{T}$ to the input and output of the encased cipher $\mathcal{E}$ does not revert the effect of $\mathcal{E}$. To analyze whether this holds, we follow a line of reasoning similar to the one reported in [8] for the security of cascade ciphers, and assume $\mathcal{E}$ to be a strong cipher, i.e., a random permutation of the input space for each $\mathtt{K}_{\mathcal{E}}$. Such an assumption is commonly maintained to be met by mathematically unbroken block ciphers. As a consequence, in the $\mathcal{TET}$ construction, for any possible value of the key $\mathtt{K}_{\mathcal{T}}$ employed by $\mathcal{T}$, the probability that $\mathcal{E}_{\mathtt{K}_{\mathcal{E}}}^{-1}=\mathcal{T}_{\mathtt{K}_{\mathcal{T}}}$ holds (for at least one value of $\mathtt{K}_{\mathcal{E}}$) is: $2^{|\mathtt{K}_{\mathcal{T}}|}\cdot\frac{2^{|\mathtt{K}_{\mathcal{E}}|}}{(2^b)!}$. In our case $|\mathtt{K}_{\mathcal{T}}|=2|\mathtt{K}_{\mathcal{E}}|+b$, and $|\mathtt{K}_{\mathcal{E}}|\geq b$ (see Figure 1(a) and Figure 1(b)), as the block cipher key typically meets or exceeds its block size. Therefore, the probability of $\mathcal{T}$ inverting the effect of $\mathcal{E}$ is $\approx\frac{2^{4b}}{(2^b)!}$, which is negligible for any typical size $b\geq 64$ of a block cipher input. As a consequence, the cascade $\mathcal{TET}$ has a security margin against exhaustive key search greater or equal than the one of the encased cipher alone.

Concerning feature **(S2)**, the crucial point is that the attacker should not be able to derive the value of $\mathtt{K}_{\mathcal{T}}$ without resorting to either the exploitation of side channel leakage or an exhaustive search over $\mathtt{K}_{\mathcal{E}}$. An attacker, knowing the value ofthe plaintext P, should derive the knowledge of both the value of the output of the first instance of $\mathcal{T}$ and the corresponding value of $\mathtt{K}_{\mathcal{T}}$ (see Figure 1(a)). Since $\mathcal{T}$ is bijective for any given $\mathtt{K}_{\mathcal{T}}$, it is possible to find at least one value for $\mathtt{K}_{\mathcal{T}}$ for any possible output of $\mathcal{T}$. Since the output of $\mathcal{T}$ is unknown to the attacker, all the key guesses relying on P alone are equally valid. In order to validate a key guess κ for $\mathtt{K}_{\mathcal{T}}$, the attacker will need to compare $\mathcal{T}_{\kappa}(P)$ with the result of the following computation on the ciphertext C: $\mathcal{E}_{\mathtt{K}_{\mathcal{E}}}^{-1}(\mathcal{T}_{\kappa}^{-1}(C))$. Under the assumption of $\mathcal{E}$ being an unbroken block cipher, the aforementioned condition can be checked for correctness only performing an exhaustive search for the value of $\mathtt{K}_{\mathcal{E}}$. A similar point on the possibility for an attacker to derive $\mathtt{K}_{\mathcal{T}}$ starting from the ciphertext C can be made simply swapping the roles of the input and output of $\mathcal{T}$. It is not possible to retrieve the value of $\mathtt{K}_{\mathcal{T}}$ without resorting to an exhaustive key search for $\mathtt{K}_{\mathcal{E}}$, or to side channel information.

While employing different keys for the two instances of $\mathcal{T}$ depicted in Figure 1(a) would still provide feature **(S2)**, there is no evident security loss in employing the same

key twice [14].

To have $\mathcal{T}$ fulfill features **(S3)** and **(S4)**, namely the need to guess at least $|\mathtt{K}_\mathcal{E}|$ bits of $\mathtt{K}_\mathcal{T}$ to compute the value of either an output bit of $\mathcal{T}$ or an `xor`-linear combination of them to perform an SCA, we designed the inner structure of the Feistel function $\mathcal{F}$ as shown in Figure 1(b) and Figure 1(c). Considering that one of the two output halves of the $\mathcal{T}$ function, namely $O^l = \mathcal{F}(I^l, \mathtt{K}_1) \oplus I^r$, is influenced by the output of a single $\mathcal{F}$ function (see Figure 1(b)), it is necessary for a single computation of $\mathcal{F}$ to fulfill features **(S3)** and **(S4)** itself. As a consequence of the Feistel structure of $\mathcal{T}$ and the use of the two unrelated keys $\mathtt{K}_1, \mathtt{K}_2$ in it (see Figure 1(b)), the remaining output half of $\mathcal{T}$, O^r will be compliant with features **(S3)** and **(S4)** too if $\mathcal{F}$ enjoys them.

To attain this, we designed $\mathcal{F}$ as a composition of three layers, applied each one to the result of the precedent (Figure 1(c)). The first layer adds, via bitwise `xor` ($\oplus$), $\frac{b}{2}$-bit sized portions of $\mathtt{K}_1$ in $\left\lceil \frac{2|\mathtt{K}_\mathcal{E}|}{b} \right\rceil + 1$ iterations (with $|\mathtt{K}_1| = |\mathtt{K}_\mathcal{E}| + \frac{b}{2}$), combining via bitwise `and` ($\wedge$) the result of each `xor` with a rotated version of itself. The reason for the insertion of the `and`-combination is that performing the `xor` addition of two key slices $\mathtt{K}_{1,i}, \mathtt{K}_{1,i+1}$ in a row would allow an attacker to consider them as a single $k_{equiv} = \mathtt{K}_{1,i} \oplus \mathtt{K}_{1,i+1}$, in turn reducing the number of effective key bits. The reason for rotating one of the operands of the `and`-combination is that not doing so would result in the `and`-combination outputting the unchanged value of its operands. The rotation coefficients ρ_i are obtained via exhaustive search, checking when features **(S3)** and **(S4)** are satisfied on the ANF representation of the output bits of $\mathcal{F}$.

Following the key addition layer, a diffusion layer, realized as a sequence of `xor` combinations of the intermediate state with a rotated copy of itself by amounts equal to 2^i, $i \in \{0, \ldots, \log_2(b) - 3\}$, is present. The reason for the rotation indexes being limited to the aforementioned values is to avoid term cancellation due to double-`xor`-additions of the same monomials. Examining the ANF of the output bits of the diffusion layer, we obtain that each single one depends on at least $|\mathtt{K}_\mathcal{E}|$ key bits. As a consequence, $\mathcal{F}$ satisfies already feature **(S3)** since predicting any single output bit of it requires guessing at least $|\mathtt{K}_\mathcal{E}|$ key bits of $\mathtt{K}_\mathcal{T}$. This implies, according to the security model in [1], a computational effort of $2^{|\mathtt{K}_\mathcal{E}|}$ to perform a side channel attack targeting one of such bits as intermediate value. Consequentially, leading a side channel attack targeting any intermediate value of the encased cipher will require a higher computational effort than obtaining its key via exhaustive search [1].

Concerning feature **(S4)**, the $\mathcal{F}$ function should be designed so that an `xor`-combination of its outputs cannot be computed without guessing less than $|\mathtt{K}_\mathcal{E}|$ bits of $\mathtt{K}_\mathcal{T}$. To this end, the final component of the $\mathcal{F}$ function (i.e., the non-linear combination in Figure 1) is a bitwise `and` of its state bits with a copy of themselves rotated by $\frac{b}{4} - 1$. We picked the rotation amount equal to $\frac{b}{4} - 1$ after checking via symbolic computation that the aforementioned value yields good results in terms of producing a significant amount of monomials which are appearing only once in the ANFs of all the output bits of $\mathcal{F}$. A viable way to confirm fulfillment of feature **(S4)**, is to check that the ANF of each output bit of $\mathcal{F}$ contains at least $|\mathtt{K}_\mathcal{E}|$ monomials, each of which involves a different bit of $\mathtt{K}_\mathcal{T}$, and never appears in the ANFs of the other output bits of $\mathcal{F}$. If the aforementioned condition holds true, there is no `xor`-combination of output bits that will make the unique monomials vanish in its result.

Obtaining a set of values of the rotation coefficients ρ_i making features **(S3)** and **(S4)** hold for the whole $\mathcal{F}$ with an encased cipher key size and block size equal to

Table 1: Comparison among SCA countermeasure strategies applied to our $\mathcal{T}$ transformation to protect AES-128

	(d, s)	AES-T		AES-S		AES-C	
		Time (μs)	**Slowdown**	**Time** (μs)	**Slowdown**	**Time** (μs)	**Slowdown**
None	$(0, 1)$	16.9	$\times$1.00	78.4	$\times$1.00	1020	$\times$1.00
$\mathcal{TET}$	$(0, 1)$	24.2	$\times$1.43	89.1	$\times$1.14	1030	$\times$1.01
TL-ISW-t	$(1, 2)$	95.2	$\times$5.63	165.2	$\times$2.11	1117	$\times$1.10
TL-ISW	$(1, 3)$	204.0	$\times$12.07	274.8	$\times$3.51	1226	$\times$1.20
TL-TI	$(1, 3)$	110.1	$\times$6.51	180.0	$\times$2.30	1131	$\times$1.11
ISW	$(1, 5)$	393.2	$\times$23.26	468.3	$\times$5.97	1416	$\times$1.39
TI	$(1, 5)$	189.3	$\times$11.20	255.5	$\times$3.26	1201	$\times$1.18
TL-ISW-t	$(2, 3)$	250.3	$\times$14.81	321.0	$\times$4.09	1272	$\times$1.25
TL-ISW	$(2, 5)$	559.6	$\times$33.11	612.1	$\times$7.81	1563	$\times$1.53
TL-TI	$(2, 5)$	322.0	$\times$18.95	387.8	$\times$4.95	1339	$\times$1.31
ISW	$(2, 9)$	2105.0	$\times$124.56	2222	$\times$28.34	3095	$\times$3.03
TL-ISW-t	$(3, 4)$	441.9	$\times$26.15	511.0	$\times$6.51	1462	$\times$1.43
TL-ISW	$(3, 7)$	1089.0	$\times$64.44	1166	$\times$14.87	2118	$\times$2.08
ISW	$(3, 13)$	3402.0	$\times$201.30	3590	$\times$45.79	4402	$\times$4.32
TL-ISW-t	$(4, 5)$	754.4	$\times$44.63	824.5	$\times$10.52	1776	$\times$1.74

$|\mathtt{K}_{\mathcal{E}}|=b=128$, $|\mathtt{K}_{\mathcal{T}}|=384$, took 1200 CPU-hours on a dual Intel Xeon E5-2630 v3, with 128 GiB of DDR4-2133. The constants found are $\rho_1=1, \rho_2=3, \rho_3=53$, and can be used for all block ciphers with the aforementioned block and key size. We note that any other configuration satisfying features **(S3)** and **(S4)** would be equally fine.

Statement 3.1 (SCA resistance of $\mathcal{TET}$)**.** *Let $\mathcal{TET}$ be a construction with a $\mathcal{T}$ transformation satisfying features **(F1)**, **(S1)-(S4)** protected with d-th order SCA countermeasures. The computational effort required to recover the encased cipher key $\mathtt{K}_{\mathcal{E}}$ is lower bounded by the minimum one between an exhaustive search for $\mathtt{K}_{\mathcal{E}}$ and a $d+1$-th order SCA against $\mathcal{T}$.*

To perform an SCA with order smaller than $d+1$ trying to retrieve a portion of $\mathtt{K}_{\mathcal{E}}$, the attacker will need to recover either a portion of the input to $\mathcal{E}$, i.e., $\mathcal{T}_{\mathtt{K}_{\mathcal{T}}}(P)$ or a portion of its output, i.e., $\mathcal{T}^{-1}_{\mathtt{K}_{\mathcal{T}}}(C)$. This is a consequence of the fact that provably secure d-th countermeasures are unconditionally secure against any SCA with order lower than $d+1$, and thus nothing can be gathered from the side channel information coming from the computation of any of the intermediate values of the protected $\mathcal{T}$ [12], leaving only the side channel information coming from the ends of the computation of $\mathcal{T}$ to be fruitfully exploited. Since $\mathcal{T}$ satisfies **(S3)** and **(S4)**, any SCA trying to predict the value of one of its output bits or a combination thereof will incur in a computational effort $O(2^{|\mathtt{K}_{\mathcal{E}}|})$, the same required for an exhaustive key search of $\mathtt{K}_{\mathcal{E}}$.

In case the attacker is able to carry out a $d+1$ order attack, it is possible for him to retrieve the value of $\mathtt{K}_{\mathcal{T}}$ extracting it in suitably sized portions (e.g., bitwise), possibly faster than an exhaustive search for the value of $\mathtt{K}_{\mathcal{E}}$. The number of measurements to perform a d-th order attack against an implementation grows exponentially with the

Table 2: Comparison among the execution time of the proposed countermeasure and the existing ones as a function of the protection level d. The results are clustered according to the implementation strategy employed for the AES-128 cipher: a single T-Table (AES-T), a single tabulated S-Box (AES-S), and a fully computational implementation (AES-C). All slowdowns are computed w.r.t. the corresponding unprotected (d=0) implementation. All absolute running times and slowdowns are provided on a Cortex-M4 based μC clocked at 84MHz save for the AES-S column marked with † from [1], which are reported from experiments running on a 1.2GHz Cortex-A9 due to the lack of publicly available source code

	AES-T		**AES-S**						**AES-C**	
	[This work]		**[This work]**		**Ref. [1]†**		**Ref. [7]**		**Ref. [20]**	
d	**Time** (ms)	**Slowdown**	**Time** (ms)	**Slowdown**	**Time** (ms)	**Slowdown**	**Time** (ms)	**Slowdown**	**Time** (ms)	**Slowdo**
0	0.01	×1.00	0.07	×1.00	0.06	×1.00	0.07	×1.00	0.3	×1.00
1	0.09	×5.63	0.16	×2.11	0.33	×5.48	88.4	×1151.2	6.6	×20.0
2	0.25	×14.81	0.32	×4.09	0.98	×16.17	217.5	×2830.5	15.6	×47.2
3	0.44	×26.15	0.51	×6.51	1.98	×32.62	400.7	×5214.7	28.7	×86.4
4	0.75	×44.63	0.82	×10.52	–	–	637.8	×8300.3	45.6	×137.6

exponent being d+1 [16]. Since the value of d is chosen by the designer, it is possible for him to choose the most fitting tradeoff between the computational overhead imposed by the SCA countermeasures and the desired security margin. As summarized in Section 2, such computational overhead grows quadratically in the number of boolean `and` to be computed, linearly in the number of `xor`s. The computation of $\mathcal{T}$ involves only a limited amount of $\frac{b}{2}$-bit wide Boolean `and`s, namely $2(2|\mathtt{K}_{\mathcal{E}}|/b) + 2$. For instance, encasing a $b = 128$ bit block cipher with b=$|\mathtt{K}_{\mathcal{E}}|$=128, requires only only 8, 64-bit wide, `and` operations for each computation of $\mathcal{T}$, resulting in a greater ease in the application of the SCA countermeasures described in Section 2 with respect to the current state of the art methods, such as the ones proposed in [1, 7, 20]. Although we do not claim the proposed solution is minimal either with respect to the amount of nonlinear operations, nor with respect to the amount of key material employed in $\mathtt{K}_{\mathcal{T}}$, the experiments in Section 4 show that significant gains can be achieved employing it.

3.3 Automated Countermeasure Instantiation

We realized the SCA protected keyed transformation $\mathcal{T}$ as a `C++11` template library providing the possibility of choosing the employed countermeasure strategy (ISW masking, tweaked ISW masking or TI, as described in Section 2) and the protection order d. Our template library providing the protected $\mathcal{T}$ takes as a parameter d, the countermeasure strategy, b and the base type of the array with which both the cipher state and the key are represented, together with the number of such elements constituting $\mathtt{K}_{\mathcal{E}}$.

To tackle the transition leakage (described in Section 2) and the possible pitfalls which may take place in the encoding phase of the countermeasures we modified the LLVM compiler toolchain allowing it to lower two opaque built-ins, `__builtin_crypto_xor`

and __builtin_crypto_and, into intrinsic operations employed by our template library to compute all the xor and and operations in $\mathcal{T}$ and the encoding/decoding phases enclosing it.

Both intrinsic instructions are lowered by the *instruction selection* pass of the LLVM backend of the desired target ISA (ARM-Thumb2 in our case) into appropriate pseudo-instructions, which specify the constraint that the destination register should be distinct from both source ones. This constraint is imposed so that it is possible, through a local transformation only, to precharge the destination register to 0 before the result is actually stored, without damaging either operand of the instruction. We note that this transformation is possible assuming that the target ISA supports three-operands instructions, such as it is the case of ARM (both classic and Thumb/Thumb2) and MIPS. We also note that no other constraint is imposed by our approach on the target ISA making it easily applicable to other ones.

Right after the *register allocation* pass has been run, we tackle the issue of protecting the transition leakage of memory operations in the functions of the source code where the keyed transformation is called. We recall that the register allocation pass may perform *spill* actions, i.e., push the contents of a register onto the *stack*, whenever the set of useful registers to store a variable are exhausted. Symmetrically, whenever the spilled value is required to compute an instruction, the value is subject to a *fill* action, loading it back into an available register. To this end, we add an extra *stack-slot* to the allocated ones which will be used to store the constant 0 at the beginning of this function. Moreover we reserve R9 as our support register to materialize the constant 0 whenever it is needed to protect spill operations; we note that such a reservation is not needed for ISAs which have a dedicated register set to 0 such as R0 in the MIPS ISA. Subsequently, the code of the function to be protected is augmented adding an extra store operation into the stack-slot dedicated to the constant 0 before each spill is performed, and an extra load operation from the 0-dedicated stack-slot into the destination register of each fill operation before the fill itself.

Finally, right before the *assembly emission* pass is run, we process all the pseudo-instructions, inserting an instruction setting the destination register rd to 0 via storing rd xor rd into it. Following the insertion of the zeroing of the destination register, we also insert, right after the computation of the pseudo-instruction, a zeroing action for each one of its operands which will be no longer used. As a last action, we change the opcode of the pseudo-instruction lowering it into the actual one of the corresponding ARM/Thumb2 instruction.

4 Experimental Evaluation

In this section we validate our approach on an ARM Cortex-M4 based μC. Our platform of choice is the STM32F407 μC, with 192kiB SRAM, 1MiB Flash on a commercial grade STM32F4 Discovery board, clocked at 84MHz. The μC is equipped with an RNG able to provide 32 bits of randomness every 2 clock cycles in our experimental setting. We employed three C implementations of the AES-128 block cipher. The first one, AES-T, relies on a single T-Table to speed up all the non-key-related operations of the AES round at the cost of a 768B increase in data memory; the second one, AES-S employs a single S-Box, and the third one, AES-C, does not rely on any tabulated nonlinear function, computing explicitly the SUBBYTES primitive, matching one of

the two approaches proposed in [1]. All the implementations were encased between two keyed $\mathcal{T}$ employing 384 bits of key material, to provide at least 128-bit equivalent security against SCA. The rotation constants $\rho_1{=}1, \rho_2{=}3, \rho_3{=}53$ of the $\mathcal{F}$ function were obtained through exhaustive search, as reported in Section 3. The implementations were compiled with our modified LLVM 3.4 compiler and release-grade (`-O3`) optimizations before being loaded on the board, producing binaries both with our transition leakage protection (marked as TL in the tables), and without. We checked that, after the templates were instantiated, and the full set of release grade optimizations acted on the code, no instrumental code from the templates was left, other than that of the desired protection. We obtained the timing results measuring the time between the assertion of a GPIO at the beginning of the execution, and its deassertion at the end sampling it with a Picoscope 5203 DSO, at a sampling frequency of 1GSa/s. All the timing measurements are averaged over 30 executions, and exhibit a sample standard deviation lower than 1% of the sample mean. Table 1 reports the result of the comparison of the execution time and code sizes obtained applying different side channel attack countermeasures to the $\mathcal{T}$ transformations in the augmented ciphers to provide protection up to $d{=}4$, together with the required number of shares s. In particular we provide results on the regular ISW masking, and its tweaked version (marked with –t) trading off extra pressure on the RNG for a lower amount of shares. We provide results employing the TI pointed out in [19] for all the degrees having available formulas.

Table 1 shows how the TL protected tweaked ISW masking yields the lowest possible overhead per protection level (i.e., value of d) on our target platform (corresponding rows are marked in gray), closely followed by the TI of [19]. It is noteworthy to mention the fact that the employed TIs requires $2d{+}1$ shares to achieve the same protection level of the tweaked ISW, and comparing its computational requirements with the ones of the regular ISW shows that TIs are able to provide the same protection level with about half of the overhead, in turn confirming their greater efficiency with respect to an ISW scheme employing the same amount of shares s. As there are currently no openly available, high-order TI instances providing d-th order security employing $d{+}1$ shares only, nor there is a formal constraint prohibiting such a construction, we deem the direction of designing such a TI a promising one to provide efficient countermeasures. Data in Table 1 provided show that, on our platform, the tradeoff offered by the tweaked ISW is favorable with respect to its regular version, as the throughput of the RNG is high enough not to penalize the tweaked implementation for requiring more random values. Indeed, the tweaked ISW protection is roughly twice less demanding than the un-tweaked one. Finally, applying our automated protection against transition leakage allows significant gains with respect to employing a higher number of shares in any of the protection schemes. The gain increases with the protection level, as an increase in the number of shares implies a quadratic growth in overhead, while our transition leakage protection has a linear cost in the number of protected instructions.

Table 2 reports the comparison of the proposed approach with the results available in open literature on applying high-order SCA countermeasures to software implementations of the AES block cipher. In particular, the results of both [7] and [20] were obtained running the `C` implementation available from [7] on our platform, taking care of replacing the call to a software PRNG with a load operation from the platform hardware RNG for the sake of a fair comparison. We report the results of [1] on its experimental platform, namely a 1.2GHz Cortex-A9 SoC (TI-OMAP4460). The re-

Table 3: Slowdowns of the fastest protected implementation of each work compared with the unprotected (d=0) AES-T (the fastest unprotected implementation)

	AES-T [This work]	**AES-S Ref. [1]†**	**AES-S Ref. [7]**	**AES-C Ref. [20]**
d	**Slowdown**	**Slowdown**	**Slowdown**	**Slowdown**
0	$\times 1.00$	$\times 3.59$	$\times 4.54$	$\times 19.63$
1	$\times 5.63$	$\times 19.72$	$\times 5234.3$	$\times 394.3$
2	$\times 14.81$	$\times 58.20$	$\times 12869.8$	$\times 928.4$
3	$\times 26.15$	$\times 117.36$	$\times 23710.1$	$\times 1698.2$
4	$\times 44.63$	–	$\times 37739.6$	$\times 2703.6$

sults in Table 2 show how our approach compares favorably in terms of performance both to the one of [7] where a software AES implementation with a single S-Box is protected performing table re-computation in a provably secure fashion, and the one of [20] where an AES implementation exploiting a computational S-Box is protected employing ISW masking and tailoring the computation of the S-Box so to be efficient with it. In both cases, our approach provides speedups greater than an order of magnitude, with growing gains when higher order implementations are considered. The comparison with the approach of [1] where provably secure countermeasures, namely the ISW masking, are applied automatically only to the first and last rounds of a computational S-Box implementation of the AES cipher is favorable: $2.11\times$ in our S-Box based implementation versus 5.48 for the case of d=1 and growing with d up to $6.5\times$ versus $32.6\times$ for the highest protection level provided by [1]. This highlights how protecting the $\mathcal{T}$ transformation is more efficient than protecting a few rounds of AES.

In Table 3 we report the comparison in terms of computation time among the best solutions available in [1, 7, 20] and ours with the unprotected AES-T implementation, which is the fastest one on our target platform. The provided data show how our solution is the one having the lowest slowdowns for all ds, even when compared against results obtained on a faster platform, such as the ones of [1]. In particular, we note that comparing our implementation against the fastest from [1], a portion of the speedup is to be be ascribed to the possibility of employing the T-Tables AES variant, which is inherently faster than the one relying on S-Boxes used in [1]. However, a direct comparison of our protected AES-S implementation with the one in [1] (see Table 2) reports speedups in the $\times 2$ (d=1) to $\times 3.8$ (d=3) range, regardless of our implementation running on a significantly less performant platform.

5 Concluding Remarks

We presented a protection strategy against SCA relying on encasing a block cipher implementation between two SCA-protected keyed transformations $\mathcal{T}$. The experimental campaign showed significant performance improvements w.r.t. alternative solutions providing the same security margin through the use of provably secure countermeasures. The performance gains are due to the lightweight nature of the keyed transformation $\mathcal{T}$, and the reduced amount of expensive-to-protect nonlinear operations in it.

Acknowledgements. This work was supported in part by the EU grant awarded for

the actions: "SafeCOP" (ECSEL Joint Undertaking 2015-RIA) Grant agreement no. 692529 and "M^{2}DC" (EU H2020 Research and Innovation Programme) Grant agreement no. 688201.

References

[1] G. Agosta, A. Barenghi, M. Maggi, and G. 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. ACM, 2013.

[2] G. Agosta, A. Barenghi, G. Pelosi, and M. 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.

[3] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F. Standaert. On the Cost of Lazy Engineering for Masked Software Implementations. In *CARDIS 2014*, volume 8968 of *LNCS*, pages 64–81. Springer, 2014.

[4] V. Banciu, E. Oswald, and C. Whitnall. Reliable Information Extraction for Single Trace Attacks. In *DATE 2015*, pages 133–138. ACM, 2015.

[5] B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. Higher-Order Threshold Implementations. In *ASIACRYPT 2014*, volume 8874 of *LNCS*, pages 326–343. Springer, 2014.

[6] M. Bucci, M. Guglielmo, R. Luzzi, and A. Trifiletti. A Power Consumption Randomization Countermeasure for DPA-Resistant Cryptographic Processors. In *PATMOS 2004*, volume 3254 of *LNCS*, pages 481–490. Springer, 2004.

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

[8] S. Even and O. Goldreich. On the Power of Cascade Ciphers. *ACM Trans. Comput. Syst.*, 3(2):108–116, 1985.

[9] S. Even and Y. Mansour. A construction of a cipher from a single pseudorandom permutation. *J. Cryptology*, 10(3):151–162, 1997.

[10] V. Grosso, G. Leurent, F. Standaert, and K. Varici. LS-Designs: Bitslice Encryption for Efficient Masked Software Implementations. In *FSE 2014*, volume 8540 of *LNCS*, pages 18–37. Springer, 2014.

[11] A. Heuser, O. Rioul, and S. Guilley. A Theoretical Study of Kolmogorov-Smirnov Distinguishers - Side-Channel Analysis vs. Differential Cryptanalysis. In *COSADE 2014*, volume 8622 of *LNCS*, pages 9–28. Springer, 2014.

[12] Y. Ishai, A. Sahai, and D. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *CRYPTO 2003*, pages 463–481, 2003.

[13] ISO/IEC JTC 1/SC 27. Information technology – Security techniques – Encryption algorithms – ISO/IEC 18033-3-2010. `http://www.iso.org`, 2015.

[14] J. Kilian and P. Rogaway. How to Protect DES Against Exhaustive Key Search (an Analysis of DESX). *J. Cryptology*, 14(1):17–35, 2001.

[15] P. C. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to Differential Power Analysis. *J. Crypt. Eng.*, 1(1):5–27, 2011.

[16] E. Prouff and M. Rivain. Masking against Side-Channel Attacks: A Formal Security Proof. In *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 142–159. Springer, 2013.

[17] M. Renauld and F. Standaert. Algebraic Side-Channel Attacks. In *Inscrypt 2009*, volume 6151 of *LNCS*, pages 393–410. Springer, 2009.

[18] M. Renauld, F. Standaert, N. Veyrat-Charvillon, D. Kamel, and D. Flandre. A Formal Study of Power Variability Issues and Side-Channel Attacks for Nanoscale Devices. In *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 109–128. Springer, 2011.

[19] O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede. Consolidating Masking Schemes. In *CRYPTO 2015*, volume 9215 of *LNCS*, pages 764–783. Springer, 2015.

[20] M. Rivain and E. Prouff. Provably Secure Higher-Order Masking of AES. In *CHES 2010*, pages 413–427, 2010.