The MEET Approach: Securing Cryptographic Embedded Software Against Side Channel Attacks

Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, *Member, IEEE*, and Michele Scandale

I. Introduction

ENSURING information security against the threat of physical implementation attacks on cryptographic subsystems of embedded devices has been an important topic for investigation by both academia and industry, for over a decade [1]–[3]. Indeed, it was proven that the physical access to an embedded device enables the recovery of sensitive information (usually, the secret key employed by a cryptographic algorithm), which is otherwise stored in a secure memory, through the use of common off-the-shelf equipment and well-established side channel attack (SCA) methodologies. To perform an SCA, an adversary observes parameters such as power consumption [2], electro-magnetic (EM) emission [4], and computing time [5] of the device, and correlates them with the computation performed to infer the secret-key value. To this end, the attacker needs to model the behavior of the device representing the information leakage via the side channel of choice. For instance, typical power consumption models are either the Hamming weight of the value being computed by a combinatorial circuit, or the Hamming distance between the values stored in a set of registers in two consecutive clock cycles. The Hamming weight model is also employed to fit the power consumption of data transfers over zero precharged memory buses [6], [7].

To defeat passive SCAs against software implementations of cryptographic primitives, the concept of nondeterministic execution has been tackled in [8] by increasing the mismatch between the stored object code and the actual operation being executed, thus, impacting on the power profile. This research direction, however, focuses on hardware modifications to the processor pipeline to introduce both random instructions writing to unused registers and extra instructions computing algebraic identities applied to a live register [8], [9]. Such an approach requires hardware modifications to protect a software primitive, effectively resulting in a reduced applicability—it is not possible to have drop-in securitization of software primitives, as there is no way to add such a protection to existing hardware parts. Another approach to SCA prevention is represented by encryption primitives implemented fully in hardware, together with the related SCA countermeasures. However, this imposes an extra cost on the target platform, which might reduce its appeal. Moreover, picking a single set of primitives to be implemented in hardware results in a reduction of the design space for the developer, and may still need a software backup in case successful attacks are lead against them [6], [10]. We therefore focus on purely software solutions, aiming at optimizing their efficiency to fit the performance constraints while avoiding the need for expensive specialized hardware. The need to take into account security and trust as an embedded system design objective, together with the growing complexity of the target devices, has spurred a trend toward automating both vulnerability detection [11], [12] and application of SCA countermeasures [13], [14] making use of compiler concepts and electronic design automation tools.

Our approach employs a series of compile-time code transformations, performed by passes implemented in the LLVM compiler framework [15], to produce an executable code that is able to change its power profile at runtime. The proposed approach employs a combination of techniques developed in the field of dynamic compilation and code specialization, tailored to suit the needs of low-end embedded systems. A similar approach for counteracting SCAs, based on dynamic compilation, was introduced in [13]. However, dynamic compilation

Manuscript received July 1, 2014; revised December 27, 2014 and February 24, 2015; accepted April 6, 2015. Date of publication May 6, 2015; date of current version July 24, 2015. This paper was recommended by Associate Editor R. Karri.

The authors are with the Department of Electronics, Information and Bioengineering, Politecnico di Milano, Milan 20133, Italy (e-mail: agosta@acm.org; alessandro.barenghi@polimi.it; gerardo.pelosi@polimi.it; michele.scandale@polimi.it).

Color versions of one or more of the figures in this paper are available online.

imposes a significant performance overhead, which makes it unsuitable for application in the context of low-end embedded systems. By contrast, code specialization [16] employs multiple copies of the same code fragment, each of which is optimized considering specific values assumed by one or more variables. This allows optimized code to be run when some variables assume useful (and usually common) values, while the original code is run in all other cases. This approach removes the cost of dynamic compilation, as the specialization is performed statically, even though the specialized code is selected at runtime. We leverage this concept, fitting it to the needs of SCA countermeasure application: rather than generating alternative code fragments with specialized functionalities, we preserve the same functionality across all alternative code fragments, and select them through a randomization policy to change the way the same computation is actually performed. The main difference from the classical code specialization approach is the size of the code fragments for which alternatives are produced. In optimization-targeted code specialization, the size of the specialized code fragment is usually fairly large (up to an entire function), to maximize the performance benefits given by optimizing some variables as constants. In our case, the fragment size is small (one or few instructions), to limit the size of the alternative code fragments which are added, and to maximize the number of alternative execution traces in the resulting output code. Our proposal enables a developer without a security background to use the automated protection framework marking with a syntax attribute the code block which should be protected, while the contribution of the SCA expert is concentrated in providing the proper alternative code fragments in the form of a human-readable configuration file for the compiler suite. The configuration file is loaded each time the compiler is invoked to produce a protected cipher implementation.

A. Contributions

In this paper, we provide a cost effective method for securing cryptographic embedded software against passive SCAs. The proposed multiple equivalent execution traces (MEETs) approach [17] reduces significantly the resource requirements to execute the protected code, in terms of both specialized hardware (none in our case) and computation time, forfeiting also the need of the code segment to be writable, which was present in [13]. In addition to this, we employ the MEET approach to solve the issue of safe recomputation of block cipher lookup tables (LUTs) [18]. As a third contribution, we tackle, in an automated fashion, the problem of preventing information leakage from memory spills, which is currently left to manual adaptation of the algorithm.

B. Organization of This Paper

In Section II, we provide a background on SCAs and current approaches for countermeasures. In Section III, we introduce the MEET approach, detailing the modifications to the LLVM compiler toolchain. In Section IV, we provide an experimental validation of the proposed approach against both power consumption an EM emissions side channels, and a performance evaluation on all the current ISO-standard block ciphers. In Section V, we review the relevant related work. Finally, in Section VI, we draw the conclusion.

II. Background

The typical passive SCA workflow is an instance of either a known plaintext attack or a known ciphertext attack, aiming at the retrieval of the secret key being employed in the cipher implementation. The attacker is assumed to know all the details of the implementation of the cipher and is able to measure an environmental parameter of his choice to derive information regarding the secret key from it. The main strength of an SCA lies in the possibility of 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. A typical attack starts by choosing an intermediate value of the cipher depending on a small key portion (usually, 8 bits) and a known quantity (usually, the plaintext or ciphertext). The side channel (e.g., the power consumption of the device) is continuously measured during the aforementioned operation, for a large set of different, randomly distributed inputs. Subsequently, the attacker tries to predict the actual power consumption of the device 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 predictions for each value taken by the portion of the secret key under attack. Each one of these predictions is compared with the measured side channel for each time instant, through the use of a statistical tool (one of the most common ones is Pearson's linear correlation coefficient), thus, revealing the actual value of the secret-key portion, as the prediction depending on it will fit best the measurements. A direct consequence of the attack scheme is that, the larger the number of secret-key bits involved in the same operation, the harder it will be to mount an SCA, as the number of hypotheses made to grow exponentially.

Countermeasures to these attacks try to remove the correlation between the values computed at the end of the algorithm and the side channel information provided by the device. The most effective strategy involves the splitting of the intermediate values into shares, which are processed independently, and recombined at the end of the computation [6], [7]. To provide an effective countermeasure, in an n-share splitting scheme, $n-1$ shares are built out of runtime generated random values, while the last one is obtained as the combination of all of them with the unprotected value. The share splitting of the values prevents the attacker from building a side channel model independent from the random values. It has been proven in [19] that, given an n-share splitting scheme, the attacker is able to build a correct side channel model, independent from the random values, only measuring at least $n+1$ operations. Such attacks are known in open literature as high-order side channel attacks [6]. There is no practical evidence of an SCA of order greater than 3, as the technical effort to successfully perform high-order attacks raises quickly with the number of shares. However, note that, the computational effort to build these models raises quickly with n if it is not possible to determine with precision the instant in which the $n+1$ measurements should be taken (as a combinatorial approach over all the measures becomes the only solution). To this end, practical solutions typically employ a combination of the share splitting approach, commonly known as masking, either with the insertion of random delays in the computation, or a shuffling of the order in which mutually

independent instructions are computed [6], [7], [20]. In this respect, we note that the MEET approach is amenable to be combined with a masking strategy to boost the security margin offered by the masking, if so is desired by the implementor. The main downside of the provably secure masking scheme in [19] is that the performance penalty introduced is significant: figures in the 20× to 400× range are common [21], thus, calling for possible alternatives to mitigate the overhead.

An important issue in masking schemes is how to maintain the masked versions of the constant LUTs which characterize the block cipher algorithm design, as the masking schemes mandate that the random shares employed to mask the LUTs should be refreshed often.

Finally, a crucial issue to provide effective side channel countermeasures, regardless of them employing masking strategies or not, is that a proper register allocation is performed [21] and no sensitive values are spilled to memory. This requirement is mandated by the spill operations providing extra information to an attacker concerning a value held in a register, effectively providing more than one useful point in time for measurement. This issue has lead to the extreme point of either advising the developer to manually inspect the generated assembly [21], which is a significant hurdle to design automation, or, to tackle the whole implementation in assembly [22], thus, having the programmer shoulder the whole burden of implementing the cipher in low level, architecture dependent code. In the following, we propose a solution solving altogether the issue of unwanted memory spills through automating the application of a masking protection strategy to all of them, and validate it practically.

III. MEET Approach

The MEET approach aims at changing the side channel profile of the application code for both its power consumption and the radiated EM emissions. This is achieved through providing a set of code fragments $C = \{I_1, \ldots, I_n\}$, each one semantically equivalent to one of the sensitive instructions presented in the original code, and altering the original code so that, at runtime, every time the sensitive fragment should be executed, an equivalent one is randomly picked from C and executed instead. A compiler extension is in charge of transforming the code at compile time to implement the above runtime behavior; a conceptual model of this workflow is depicted in Fig. 1. The output of the modified compiler pipeline is a binary file which will be run on the unmodified embedded platform for which the compiler emits code. The modified binary can be seen as split in two portions: the instructions deemed not sensitive against attacks, which are left unmodified by the process, and the sensitive ones, which are not emitted. In place of the sensitive instructions, jump instructions toward the secure code fragments are emitted. The sensitive instructions can either be identified automatically, through the methodology proposed in [11], or the whole code block marked syntactically by the programmer can be considered sensitive, and thus subject to instruction replacement. In this paper, we employed the methodology described in [11], setting a threshold for the instruction resistance equal to the key length of the cipher. This in turn implies that all the instructions for which leading a side channel attack is less expensive than an exhaustive key search are protected. The secure code portion, shown in Fig. 1 (right)

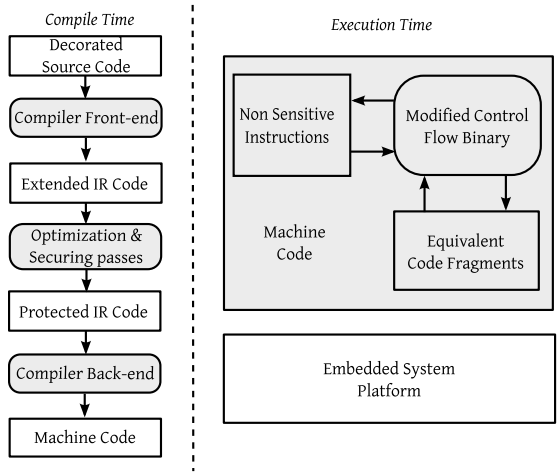

Fig. 1. Conceptual model of the proposed software architecture. Left: the typical compiler pipeline is depicted, with the portions where our extensions are inserted marked in gray. Right: a conceptual summary of the MEET protected executable code is provided.

can be thought of as split into two parts: a set of randomized control flow constructs and the equivalent code fragments. The randomized control flow is built in such a way that a sequence of instructions semantically equivalent to the original cipher is executed every time, while changing the actual specialized code fragments being used to do so.

In the rest of this section, we will first introduce the structure of the compiler modifications supporting the MEET code transformation, together with the passes which are actually performing it (Section III-A). We will then provide a description of the specification language through which the alternative code fragments to be employed in place of the sensitive operations are supplied (Section III-B). Finally, we will describe the special strategy required to secure `load` operations from LUTs (Section III-C), and the backend pass which provides the protection on the spill/fill memory actions (Section III-D).

A. Compiler Extension

We now detail the transformation flow of the security critical code operated by the proposed compiler extension, as depicted in Fig. 2. Our compiler extension is split into three portions: one concerns the syntactic support which is added to the compiler front end, the core one which resides in the compiler middle end, tackling the code transformation according to the MEET approach. Finally, the last portion tackles the memory spill protection issues in the compiler back end. The core of the code transformation is implemented as a set of function-level passes acting on the intermediate representation (IR) of the target cryptographic routine manipulated by the LLVM compiler toolchain. The LLVM IR is expressed in the form of a control-flow graph (CFG), of which we recall the definition.

Definition 1 (CFG): A CFG is a directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with one node $i \in \mathcal{V}$ for each statement of the function ($\texttt{stat}_i$), augmented with two additional nodes i_{start} and i_{end}. An edge $(i, i') \in \mathcal{E}$ is added if the statement $\texttt{stat}_{i'}$ is executed immediately after the statement $\texttt{stat}_i$. In addition, an edge is present between the i_{start} and first statement, and between each node preceding an exit point of the function and i_{end}.

The statements in the CFG are expressed in static single assignment form, i.e., every variable is statically defined

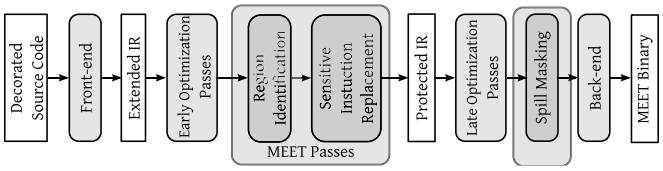

Fig. 2. Modifications to the LLVM compiler toolchain to apply the MEET approach. The boxes highlighted in gray denote modified passes, while the highlighted boxes mark the added passes.

only once. The CFG definition is easily extended to graphs where each node is a basic block.

Definition 2 (Basic Block): A basic block is defined as a connected CFG sub-graph, where each node has a single incoming edge and a single outgoing edge, save for the first and the last one. The first node has either zero or more than one incoming edge, while the last node has either zero or more than one outgoing edge.

1) Front End: To provide the means to the developer to specify which part of the code should be manipulated by our extension, the front end of the LLVM toolchain, `clang`, has been extended to support two custom keywords. The `__meet` keyword is prefixed to a `C` code block (i.e., a sequence of statements wrapped by curly braces) to mark it for protection. The `__sbox` keyword decorates an array of constants usually accessed by the cipher through a key-dependent value (e.g., the AES SBOX). This allows to identify the sensitive operations (`loads`) acting on such variables and to apply a proper countermeasure. The `__meet` keyword is lowered into the LLVM IR as a MEET code region, which is defined as follows.

Definition 3 (MEET Code Region): A code region is defined as the portion of the CFG in the LLVM IR, corresponding to a code block, and delimited by custom entry- and exit-marker. An entry-marker is placed in correspondence of each entry point of the region, be it a `C` language `label` within the code block or the first statement of it. An exit-marker is placed in correspondence of each `goto` statement pointing to a label outside the marked block, each `return` statement within the marked code block, as well as the last statement of it.

Each marker of an MEET code region is implemented as a single basic-block wide loop, in which the back edge to the beginning of the basic block is controlled by a condition computed by a call to a custom opaque compiler intrinsic. In the subsequent compiler passes, the call to an opaque intrinsic prevents the region markers to be deleted, as the side effects of its execution cannot be predicted at compile time. The loop structure of each marker isolates it from instructions in other basic blocks, realizing a control-flow barrier across which instructions cannot be moved by nonloop-based optimizations.

2) Optimization Passes: The optimizing passes on the IR code have been modified to take into consideration the constraints imposed by the code regions. Indeed, the main issue lies in preventing the loop-invariant code motion of instructions lying within a code region outside of it. In particular, the code hoisting actions in the optimizer check if an instruction is going to cross a region boundary before actually moving it (e.g., when triggered by a loop-invariant instruction included in a region contained in a loop body). This effectively avoids the extraction of sensitive instructions from the code region to be protected. Symmetrically, code sinking optimizations may lead to the inclusion in the region of nonsensitive instructions. Although this does not represent an explicit security threat, it will reduce performances, and is thus prevented. Moreover, the induction variable simplification pass has been modified so to prevent the motion of preheader statements, which are not found to be used within the loop body, and can be moved from the preheader into the loop exit block, provided it is a single one. This code motion may cross a region boundary and is thus effectively prevented.

To allow a selective application of different types of optimizations to different portions of the code region, the opaque intrinsic defined to implement it takes one meta-data argument, which is used to uniquely identify the code region itself, as well as to specify which behavior a transformation pass should take in the code region. For instance, in a suitably marked code region, the instruction combine pass must refrain from performing the optimization as it might remove the effect of the protection by simplifying the IR code. Referring to Fig. 2, there are two optimization stages: the first is run before the application of the new MEET pass, while the second acts on the protected IR code. This strategy was adopted as some optimizations may be advantageously run twice (e.g., inlining), once before and after the protection is applied.

3) Region Identification: After the application of the optimizing steps, some of the IR instructions included in the marked code regions may be optimized out or otherwise modified. To identify each instruction as belonging to a specific code region, this compiler pass builds, for each code region, a data structure to store the instructions that belong to it. To this end, this pass exploits the concept of k-vertex dominance [23] of the entry/exit markers of the code region.

Definition 4 (k-Dominance Relation): Given $v \in \mathcal{V}$ and a set $\mathcal{S} \subseteq \mathcal{V}$ of CFG nodes with cardinality $|\mathcal{S}| = k$, the set $\mathcal{S}$ is said to k-vertex dominate v if: 1) every path from the start node to v must go through at least one of the nodes in $\mathcal{S}$ and 2) for each $w \in \mathcal{S}$ there is at least one path from the `start` node of the CFG to vertex v, which contains w but not any other vertex in $\mathcal{S}$.

Considering a code region, we note that the entry (respectively, exit) markers are, by construction, k-vertex dominators (respectively, post-dominators) of the instructions belonging to it. The region identification traverses the CFG of the cryptographic primitive, to collect the markers of every region. For each pair of entry and exit marker sets of a certain code region, the compiler pass backward visits the CFG starting from the exit markers until it meets either a corresponding entry marker or an already visited node. During the visit, the compiler pass updates a data structure for each code region with nodes identified as included in it. The aforementioned dominance property guarantees the correctness of such a procedure as every path that goes back from an exit marker to the `start` node of the CFG meets at least one entry marker. A properly implemented cipher will be characterized by both a single entry and a single exit point for every code region to avoid nonconstant execution paths which would pave the way for timing analysis vulnerabilities.

4) Sensitive Instructions Replacement: All the sensitive instructions contained in a code region are recognized by the first stage of this pass, matching each of them against the content of a description file including the set

of semantically equivalent secure alternatives (called code tiles—see Section III-B), which are specialized to match the behavior of the original instruction in terms of memory address, immediate values and registers used. The sensitive instructions are subsequently processed by the second stage of this pass, which inserts single-instruction level code specializations to thwart consumption-based SCAs. Indeed, each security sensitive instruction is replaced with a selection construct, akin to the common `switch-case` of the `C` language, driven by a runtime generated random number, which picks one out of a set of code fragments, semantically equivalent to the original instruction. As the original sensitive instruction outcome is computed by means of a different operation sequence, the side channel profile of the performed computation will vary at each execution. The change in the computation is steered by a random number generator (RNG), resulting in a nonpredictable mismatch between the side channel behavior model devised by the attacker and the actual measurement of the device. The construction of a correct model by the attacker is effectively hindered by both continuous changing in the executed code and the time shifting induced by the different length of the computational alternatives for the same original instruction. Should an hardware RNG not be available on the target platform, it is possible to cope with the overhead of a software RNG through filling a buffer with its outputs, and reusing them for a small number of computations of the cryptographic primitives, similarly to the approach of [13].

We note that the first sensitive instruction does not benefit by the inherent time shifting induced by the ones being computed before it. However, the application of a provably secure masking strategy [19] to compute the first instruction, and the substitution of each of the replacement instructions employing the MEET approach, can easily provide security also in this case. Indeed, the lack of time shift affecting the first sensitive instruction can be effectively compensated with a very low overhead. An analogous approach should be followed to protect the last sensitive instruction of the cipher, as it is possible with digital storage oscilloscopes to perform pretrigger capturing of the power consumption, thus effectively allowing an attacker to synchronize the measurements on the end of the computation. This measurement setup removes the time shifting from the last sensitive instruction execution, calling the same approach used for the first instruction.

Concerning attack strategies, we also take into account the outcomes of an *a posteriori* executed instruction classification, starting from the side channel measurements. We report that employing a machine learning approach to classify the computed instructions according to their side channel leakage has been conducted with unsatisfactory results as mentioned in [24]. In particular, it has been proven that an attacker in possess of the whole value change dump for each clock cycle of a computation of a known netlist 6502 μC implementation is unable to distinguish computational instructions when given their leakage on the power consumption side channel. A modeling approach based on the principal component analysis of the power consumption traces of an 8-bit microcontroller [25] still yields computational instruction recognition rates below around 40%–60% for both arithmetic Boolean bitwise operations and branch instructions.

B. MEET Code Tile Definition

As detailed above, the code transformation flow takes as input the specification of a set of code tiles (i.e., sets of instructions sequences, performing the same computation as a given one) to perform the instruction replacement pass. Multiple equivalent tiles for a sensitive instruction must be specified; it is also possible to include tiles built out of common side channel countermeasure strategies such as hiding and masking [6], [7], [19]. The tiles are expressed in a normalized form which involves a uniform renaming of the registers, and considering a single meta-value for immediate constants. This allows the security specialist, who needs to encode them, to factor the instruction alternatives differing only by a renaming of the registers, or the value of an immediate constant, expressing them in a friendly way. The MEET toolchain is able to parse tiles specified in a text file according to the following extended backus-naur form grammar:

$$
\begin{aligned}
S &\rightarrow R^{+} \\
R &\rightarrow I\ '\{'\ T^{+}\ '\}' \\
T &\rightarrow '['\ I\ (',\ I)^{*}\ ']' \\
I &\rightarrow V = O_b\ A\ A \mid V = O_u\ A \\
O_u &\rightarrow \texttt{not} \mid \texttt{neg} \\
O_b &\rightarrow \texttt{and}|\texttt{or}|\texttt{xor}|\texttt{add}|\texttt{sub}|\texttt{mul}|\texttt{shl}|\texttt{shr}|\texttt{ashr} \\
A &\rightarrow V \mid [\,'-'\,]['0'-'9']^{+} \mid \texttt{rand}['0'-'9']^{+} \\
A &\rightarrow \texttt{result\_size} \mid \texttt{const}['0'-'9']^{+} \\
V &\rightarrow '\%'['0'-'9']^{+}.
\end{aligned}
$$

The code tiles are represented as a list of pairs (R): the first element of each pair is a sensitive normalized instruction, while the second one is written down as one or more lists of tiles ($'\{'\ T^{+}\ '\}'$). Each tile (T) is constituted by (nonsensitive) normalized instructions (I) enclosed by square brackets. The normalized instructions constituting a tile are represented as the assignment of the result of either a unary (O_u) or a binary (O_b) operation in prefix notation. Each unary/binary instruction must have a destination virtual register (V) and may take as arguments (A): registers, constants, the size of result of the instruction, or placeholders for run-time picked random values. The format tackles the specification of code tiles for all the computational instructions, whereas `load` and `store` operations are managed separately.

Concerning `store` operations, we note that it should never happen that a sensitive value is stored in memory as-is, as it may leak information on the stored value directly. Typically, the extra `store` operations are added to handle values being spilled from the registers to memory due to excessive register pressure. Since tackling register spilling issues must be done after the register allocation pass of the compiler backend, we solved the issue through a dedicated pass which is described in Section III-D. Note that, the `store` operations saving the final computation output back to memory do not need any protection, as they do not leak any additional information with respect to the knowledge of their outcome values.

In block ciphers, the `load` operations, not resulting from the need of a fill action after a previous spill, are performed either from fixed LUTs (e.g., S boxes) or concern the initial fetch of input and key material values. Concerning the `load` operations of the key material, it is not possible to exploit them

to perform effective side channel analysis, as their side channel profile does not vary depending on the input to the cipher. The `load` operations involving input data are not a concern either, as the typical SCA scenario considers an attacker who knows both cipher inputs and outputs. The remaining `load` operations, i.e., the ones performed from LUTs, are typically indexed with values depending from both cipher input and a portion of the secret-key material. They represent an easy information leakage spot for software cipher implementations, and are dealt as follows.

C. Load Replacement and Lookup Tables

Memory `load` operations cannot be rewritten in terms of semantically equivalent instruction sequences with a different side channel leakage model, as the value transits in clear on the memory bus. To counteract the side channel leakage arising from memory operations, we adopt the share splitting technique (also known as masking), an effective approach to prevent information leakage [6], [7]. For the sake of clarity, we will describe a two-share splitting approach, however, it is straightforward to generalize this to an arbitrary number of shares. The core idea of the masking for `load` operations is to represent each value held in memory as two shares, where one is a randomly chosen value, and the other is the result of the bitwise `xor` between the random value and the original one. `load` operations are translated into pairs of `load`s, followed by their recombination via `xor` in the CPU registers.

The LUT masking approach also mandates that the random values used to mask the ones in the LUTs should be periodically changed, with the ideal case being before any further access to the same cell is made. Particular care should be taken in performing this action, as it should be completely independent from the data being processed in the cipher. The typical strategy to refresh the random values involves adding via `xor` a fresh nonce to all the memory stored shares, via a simple loop sweeping the whole share-split LUTs [7]. However, this strategy has been proven to be vulnerable to the retrieval of the fresh nonce via passive SCA, as the loop-based update strategy provides ground for the attacker to set-up a correct model of the refreshing procedure, and derive the random values even from a single refresh action [18]. Currently, the only alternative to prevent this is to compute the contents of the LUTs as Boolean nonlinear functions, with a consequent significant performance penalty, whereas providing a secure strategy to refresh the random values is currently an open problem [18].

1) MEET Mask Refreshing Strategy: We propose a novel mask refreshing strategy based on a special MEET tile, which overcomes the attack strategy of [18], exploiting the multiple execution paths offered by the MEET approach. The core idea of the MEET-based mask refreshing strategy is to exploit the regular `load` operations of the cipher, retaining both loaded shares in the registers, so they can be refreshed and stored back in the LUT at a later time. By decoupling the access to a table cell from its refresh, we prevent the attacker from correlating a `load` with the corresponding refreshing `store`. The implementation is exemplified by the C-like pseudocode in Fig. 3. For the purpose of this example, consider all scalar variables as held in the CPU registers. Let `v1` and `v2` be a pair of shares from split table `Box`, and `iold` their table index. `v1` and `v2` are retained in the register so they can be refreshed. Every time `Box` is accessed, the shares `v1` and `v2`

```
unsigned v1 = BoxShare1[init]; // Initial shares
unsigned v2 = BoxShare2[init]; // to be refreshed and
unsigned iold = init;          // their table index.
...
unsigned l1 = BoxShare1[icur]; // Value shares loaded
unsigned l2 = BoxShare2[icur]; // from lookup table.
unsigned r = rand() >> (sizeof(unsigned)-1); // arith. shift
v1 = l1 & r | v1 & ~r;         // Use r to select between
v2 = l2 & r | v2 & ~r;         // old and new value shares.
iold = icur & r | iold & ~r;   // Update the table index.
unsigned mask = rand();        // Get a random mask.
BoxShare1[iold] = v1 ^ mask;   // Refresh table masking
BoxShare2[iold] = v2 ^ mask;   // for both shares.
```

Fig. 3. Example of LUT mask refreshing.

may be replaced with the shares `l1` and `l2` (which have just been loaded) with a 50% probability, depending on a random bit. To perform this action in a constant time and without any branch operation, the shares `v1` and `v2` and the index `iold` are updated using the 2-to-1 multiplexer Boolean formula (i.e., $(\alpha \wedge \sigma) \vee (\beta \wedge \neg\sigma)$), using as the selector a register-size value `r` filled with either all zeros or all ones depending on the random bit. Finally, the masks of the shares held in `v1` and `v2` are refreshed by `xor`-ing them with the nonce `mask` obtained from the RNG, and stored back to `Box`.

The `store` operations may be executed with a chosen frequency (up to once per LUT access) allowing a tradeoff between security and efficiency (as more frequent mask refresh actions imply a faster LUT refresh) and performance (as the extra `store` actions decrease it). Note that, the only constraint in inserting mask-refreshing `stores` is that at least one `load` operation should lie between them, as two consecutive `stores` would otherwise refresh the same masked value, with no practical effect, and a performance penalty.

Provided the RNG yields uniformly distributed random values, the adversary is no longer able to understand which table value is being refreshed with each `store` operation, thus, thwarting possible attacks aiming at recovering the involved random values [18].

Considering the position of the cells in the LUT as a set of distinct objects (e.g., 256 for the AES S-box), the accesses performed by each `load` operation can be modeled as a sampling, with replacement, from such a set. Indeed, the access pattern to the LUTs is uniformly distributed by virtue of the block cipher design, as it depends on the cipher input and the secret-key values. Let A_n denote the random variable defined to be the number of accesses required to fetch all n table cells. Let X_i denote the random variable with geometric distribution representing the access to the ith cell of the table, after $(i-1)$ different cells have been independently accessed. The geometric variable X_i, with parameter $p_i = (n-i/n)$, $i \in \{0, 1, \ldots, n-1\}$, has probability mass function $\Pr(X_i = x) = (1-p_i)^{x-1} p_i$. Therefore, the expected value of $A_n = \sum_{i=0}^{n-1} X_i$ is computed as $\mathbb{E}(A_n) = \sum_{i=0}^{n-1} \mathbb{E}(X_i) = \sum_{i=0}^{n-1} 1/p_i = nH_n$, where $H_n = \log n + \gamma + 1/2n + O(1/n^2)$, with Euler's constant $\gamma \approx 0.577$. Thus, on average, the cipher computation will access all the cells of the LUT in approximately $n \log n$ `load` operations.

The mask-refreshing mechanism at the core of the countermeasure will then update the last accessed LUT cell depending on the value of the random bit `r` extracted after each `load` instruction. Let N be the number of `store` operations to be executed, and let Y be a binomial distributed random variable modeling the update of different cells of the masked table.

The binomial variable Y, with parameter $p = 1/2$ (by construction), has probability mass function $\Pr(Y = y) = \binom{N}{y} p^y (1-p)^{N-y}$ and expected value $\mathbb{E}(Y) = Np$. The average number of `store` operations required to be executed for refreshing the whole LUT is thus given by $N = (\mathbb{E}(A_n)/p) \approx 2n \log n$.

The effectiveness of the countermeasure can be appreciated considering the case of the reference AES-128 implementation with a 256-entry S-box. Employing the notation introduced above, in this case we have $n = 256$ and $N = 2048$. A single run of this cipher performs 160 LUT accesses, thus, to update all the masks of the protected S-box an average of $N/160 \approx 13$ runs needs to be executed. Willing to retrieve a single nonrefreshed mask, an attacker will be able to exploit at most $N/n = 8$ side channel samples, too few for any possible approach.

We note that, to avoid attacks exploiting a stationary cold-boot state of the targeted device, the random masks of the protected LUTs must be refreshed at each cold-boot procedure. This has no practical impact on the usability of the device, as such a procedure is shorter than the typical bootstrap time of a microcontroller.

D. Management of Register Spills

The register allocation pass of the compiler backend tackles the issue of transforming the infinite register IR, after the instruction selection pass, into an IR taking into account the limited number of registers of the underlying architecture. As the number of registers is finite, the register allocation pass may decide to spill a live value, i.e., a value which will be used in future operations, into memory, so as to free up temporarily a register. Once the register becomes available again, the aforementioned value will be used to fill it back. In addition to the performance penalty of such operations, the spill–fill actions represent also a possible cause of information leakage via side channel, as additional instructions manipulating potentially sensitive values are added. We note that, despite software implementations strive to minimize spills, they are a natural occurrence on architectures with a small number of registers. Willing to provide a sound side channel protection, we took into account the protection of spill and fill operations, employing a masking strategy. In particular, the information leakage is prevented employing a random value to mask the original value to be stored via a `xor`. To this end, we reserve a register, instructing the register allocator so that it is never used for other purposes. In particular, our implementation prevents the register allocator from using the callee-saved `r9` register of the ARM architecture during the allocation action on the function where the MEET protected code resides. Employing a callee-saved register for our purposes guarantees that no alterations to the semantics of the callers of the protected function will be made. Exploiting the reserved register, it is possible to protect the spill and fill operations as follows. A protected spill operation loads a random value from the platform-available RNG into the reserved register, combines the loaded value with the value to be spilled, and stores both of them on the stack. A protected fill operation will load the protected value into the target register and its corresponding random mask into the reserved register, removing the mask from the protected value via `xor`.

The net effect of the strategy is to turn any spill–fill operation pair into a first order masked one, effectively protecting it from attacks. We note that, the same strategy can be employed to apply a higher-order Boolean masking to the value still requiring only a single register. To this end, it is sufficient to store a random mask on the stack right after it has been combined with the value to be protected, freeing up the reserved register for the next random mask to be generated and combined. The fill action for a higher-order Boolean masking can be performed by loading the masked value into the target register and the masks, one-by-one, into the reserved register recombining them with the protected value before the next mask is loaded. Particular care should be taken to precharge with a fresh random value the reserved register, before the new mask is loaded: failure to do so may provide the attacker with a side channel value related to a mask pair, thus, possibly lowering the order of the attack required to breach the protection by one [21]. We note that, combining the masks between them and using their combination to unmask the protected value effectively breaks the protection effectiveness, and should thus be avoided. Finally, we note that, the spill protection approach can be used both with Boolean (i.e., `xor`-based masking) and with additive (i.e., `add` based) masking, depending on the protected algorithm.

IV. Experimental Evaluation

In this section, we will be presenting the results of the experimental validation of the effectiveness and efficiency of the MEET approach. In particular, we will provide a practical validation of the MEET effectiveness in protecting an AES implementation against both EM and power consumption side channel attacks, and report the performance and code size results on all the block ciphers standardized by the ISO/IEC committee, i.e., the ones encompassed by ISO-18033-3 [26] (general purpose block ciphers), and ISO-29192-2 [27] (lightweight block ciphers).

A. Side Channel Analysis Evaluation

Our target architecture for the experimental evaluation is the ARM Cortex-M, which offers 16 32-bit registers, of which 12 are left available for general purpose use by the ARM Thumb2 ABI, and endowed with a zero-cycle penalty barrel shifter, typical of ARM architectures. The chosen platform is an STM32F407 microcontroller clocked at 168 MHz and endowed with 192 kiB SRAM and 1 MiB flash memory, mounted on the STM32F4DISCOVERY commercial development board by STMicroelectronics [28]. Both SRAM and flash memories are connected to the Cortex-M core via an AHB bus crossbar matrix, thus, implying that `load`/`store` operations toward the SRAM and `load` operations from the flash memory will require more than a single clock cycle. We assessed the resistance against SCAs based on both measuring the near-field EM emanations and power consumption of the STM32F407 microcontroller, providing a setup aiming at being the most advantageous possible to the attacker. To this end, the power consumption of the device was measured at the measuring point provided by the development board, after removing all the input voltage smoothing capacitors, so as to avoid unwanted low-pass filtering effects. The EM emissions were measured by means of a custom-made H-field loop probe (1 cm loop of 50 Ω coax cable with a 3 mm central gap) placed directly on the chip package. The flash

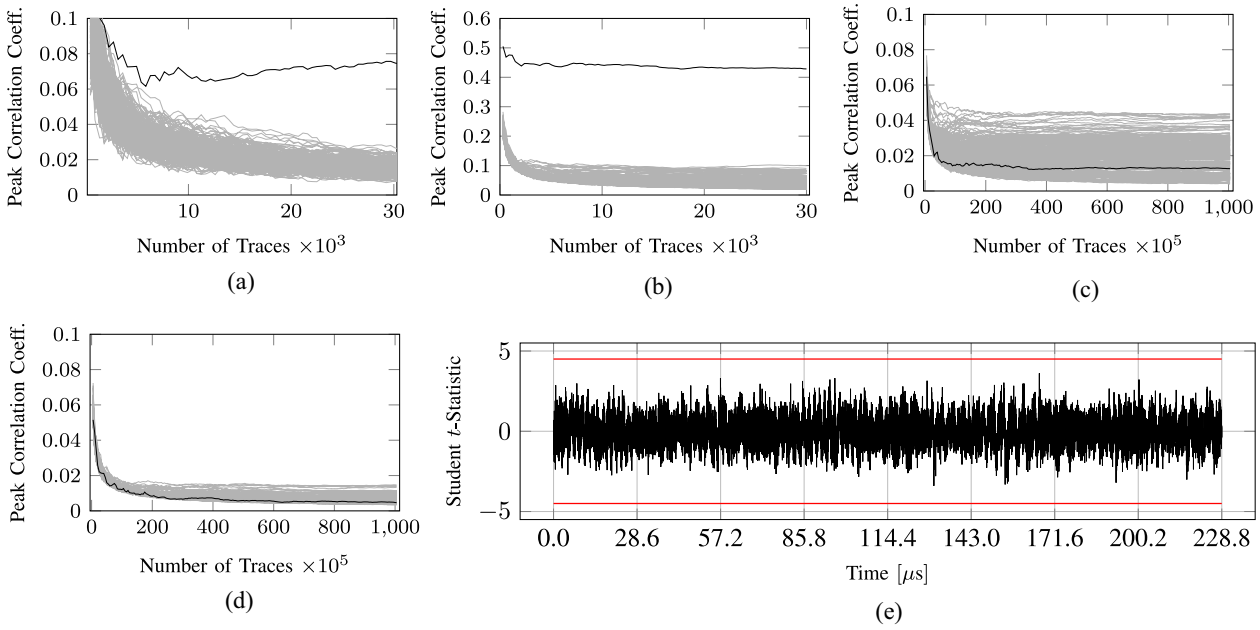

Fig. 4. Effects of the MEET protection on power and EM attacks against a reference software implementation of AES-128 on the target μC. (a) to (d) Correlation index with respect to the number of measurements of the side channel (traces). Each line depicts the evolution of the sample correlation coefficient for a key hypothesis (the correct one is highlighted in black). (a) Power attack on the unprotected code. (b) EM attack on the unprotected code. (c) Power attack on the MEET-protected code. (d) EM attack on the MEET-protected code. (e) Results of a non-specific *t*-test performed with two sets of 50 M MEET protected EM traces each, with the maximum allowed value for the *t*-statistic for the test to accept the null hypothesis highlighted in red.

memory cache embedded in the STM32F407 was disabled during the side channel measurements to enhance the time alignment, thus, providing further advantages to the attacker model. Side channel measurements were collected by means of a Picoscope 5203, sampling at 500 M samples/s, and providing accurate triggering through employing a general purpose input-output (GPIO) of the device, asserting the signal on the pin right before the encryption started and deasserting it as soon as it finished. The measured signal was amplified by means of an Agilent INA-10386 wide band amplifier, providing a 26 dB amplification up to 1.5 GHz, before being fed into the oscilloscope. To assess the level of vulnerability of the unprotected implementation on our test platform, we collected 30 K measurements with different, randomly generated plaintexts, without performing any averaging, as the noise level was low enough to allow effective attacks. To provide a sound validation margin for the automatically protected implementation, we collected 100 M measurements for both EM and power side channels with the same experimental setups.

We chose, as the security testbed for our experiments, a reference AES implementation, based on a single *S*-box, which we have verified to be characterized by a significant amount of spill actions, thus, providing an advantageous situation for the attacker. We note that, given the nature of the SCA, similar results are obtained on different block ciphers. All the object files were obtained through compiling the source `C` code into assembly employing the LLVM internal assembler and linking them with the GNU linker, due to the need for the support for explicit executable-section mapping required on the microcontroller via linker scripts. All the binaries were obtained with deployment-grade optimizations enabled (`-O3` option) and loaded on the target platform.

Fig. 4 shows the results of both differential power analysis [Fig. 4(a)] and differential EM emission [Fig. 4(b)] attacks on the target platform, employing Pearson's linear correlation coefficient as the statistical tool of choice to derive the correct cryptographic key hypothesis. The attack was performed by employing the Hamming Weight of the value loaded from the *S*-box during the first cipher round as both model for the power consumption and associated EM emissions of the computation. The partial sub-key hypothesis is thus made on a single byte of the key employed by the first ADDROUNDKEY. Fig. 4(a) and (b) shows that the EM emission attack is more successful than the power attack, as it only requires about 500 measurements instead of 4000 to distinguish the correct key. In addition, an EM leakage has a higher-absolute value for the sample correlation (0.42 versus 0.07): such results are to be expected as the EM leakage is known to be less affected by interferences from noncomputing portions of the device (e.g., integrated peripherals) [29]. Fig. 4(c) and (d) reports the results of the attack performed against the MEET protected implementation, showing that MEET effectively protects the cipher, as even employing 100M measurements the correlation coefficient of the correct key is still lower than many wrong key hypotheses (25×10^3 and 200×10^3 more traces than the ones required to lead a power or EM emission attack, respectively).

Such an increase in the number of measurements provides a sound practical security margin, as it is rather a commonplace to either actuate key refreshment strategies periodically so as to avoid the attacker gathering a very large amount of measurements with the same secret key, or to include a hardware counter which limits the effective lifetime of the device in terms of number of encryptions (e.g., ATM smart cards).

Similar attacks have been conducted by employing the Hamming weight of the output of the `xor` operations of the first and last ADDROUNDKEY, and of the last SUBBYTES reporting similar results, albeit with lower correlation values for the unprotected implementations, and the Hamming

TABLE I
ENCRYPTION SPEED AND THROUGHPUT FOR THE SET OF ISO-18033-3 AND ISO-29192-2 BLOCK CIPHERS

Reference	Cipher	Block Size	Unprotected Time [μs]	Unprotected Throughput [kiB/s]	Unprotected No. instr.	Unprotected No. spills	Unprotected IPC	Fully Protected Time [μs]	Fully Protected Throughput [kiB/s]	Fully Protected Avg. No. instr.	Fully Protected Avg. No. spills	Fully Protected Avg. IPC
ISO-18033-3 Wide-block [26]	AES-S	128	28.26	552.90	2670	696	0.56	228.80	68.29	19838	1595	0.51
	AES-T	128	11.89	1314.12	620	0	0.31	80.33	194.51	7631	477	0.56
	Camellia	128	11.85	1318.56	961	4	0.48	106.53	146.67	9290	266	0.51
	SEED	128	12.79	1221.65	775	10	0.36	135.30	115.48	13357	1276	0.58
ISO-18033-3 Narrow-block [26]	3DES2	64	29.99	260.50	1896	80	0.37	258.30	30.24	26372	3049	0.60
	CAST128	64	5.10	1531.86	319	0	0.37	44.74	174.62	4644	431	0.61
	HIGHT	64	16.08	485.85	2295	391	0.84	95.32	81.96	8868	362	0.55
	MISTY1	64	7.56	1033.39	596	116	0.46	76.37	102.29	6638	463	0.51
ISO-29192-2 Lightweight [27]	Clefia	128	36.80	424.59	4317	747	0.69	234.10	66.74	20569	1444	0.52
	PRESENT	64	77.08	101.35	9431	1034	0.72	214.90	36.35	24547	1922	0.67
[31]	XTEA	64	3.02	2586.92	482	0	0.95	18.17	429.96	1952	16	0.63

distance between the aforementioned values. All these attacks did not report a successful key extraction.

As a further validation of the effectiveness of our side-channel protection approach, we applied to the MEET-protected code the testing methodology and the related nonspecific statistical *t*-test presented in [30]. The mentioned test employs a Student's *t*-test for two sets of samples extracted from two random variables with possibly different variances (also known as Welch's *t*-test) to evaluate whether the mean value of the power consumption of the device computing a cipher with a variable input and a fixed secret key, is the same of the computation with both secret key and input fixed. This test is repeated for each time instant of the cipher computation. If the statistical test leads to accept the hypothesis that the mean values are the same (null hypothesis), this provides a sufficient condition for the device to be side-channel resistant against any nonprofiled first order attack, as changing the input does not induce differences in the device behavior. For typical evaluations [30], if the *t*-statistic is well within the interval $-4.5 < t < 4.5$, the test allows to claim that the implementation is protected with confidence $>99.95\%$.

Following the guidelines in [30], the test was conducted by collecting a set of measurements where the input was kept fixed to the same value for half of them, and randomly changed for the other half. As advised in [30] to compensate the effect of slow drifting errors due to the significant amount of time required to acquire the data, the collection of the measurements with a fixed plaintext was interleaved with the collection of the ones with a random one. The whole set of collected data should be split into two groups, to yield the two populations for the *t*-test. The criteria for splitting the acquired data, can either cluster all the fixed-plaintext measurements in a group, leaving the remaining ones as the other group, or split the measures according to the intermediate values of their computation [30]. However, the strategies are substantially equivalent, as the point of the test is proving that providing randomly distributed inputs leads to collect measurements (traces) distributed in the same fashion as the ones collected with a repetition of fixed inputs. Fig. 4(e) reports the results of performing the nonspecific *t*-test with two populations of 50 M EM traces each, obtained by splitting 100 M traces with the fixed-random plaintext strategy. As it can be noticed, for the whole duration of the fully MEET protected AES code, the obtained values of the *t*-statistic lie in the $[-4.5, 4.5]$ range, thus, leading to the acceptance of the null hypothesis. Similar results were achieved by performing the test on power traces and changing the splitting policy, and are omitted due to space constraints.

B. Performance Measurements Technique

We will now describe how the performance measurements, of which the results are reported in Table I, were obtained. First of all, we obtained the encryption time for all the algorithms measuring the time during which the GPIO signaling the beginning and end of the encryption was held high. The timing measurements were performed via the same Picoscope 5203, sampling at 500 M samples/s, thus, providing a timing precision of 2 ns. The computation of the cipher throughput was obtained by taking into account the block size of the cipher, which may be either 64- or 128-bit.

To provide an accurate measurement of the amount of spill/fill actions, both total number of spills/fills present in the encryption function, and the average spill/fill count per execution of the encryption function were measured. The former quantity was derived by counting how many memory operations were marked as acting on a spill slot, iterating over the compiler internal representation of the function right before the assembly printing phase. The latter quantity was derived by a custom analysis pass on the encryption function, also executed just before the assembly printing phase. The basic idea is to compute the average number of spills/fills on every execution path of the function. Assuming that the implementation of a block cipher is fully unrolled, the analysis over the protected function must handle only the branch divergences induced in the CFG by sensitive instruction replacement. Note that, the divergences are only local, i.e., a single-entry single-exit (SISO) region, thus, we can logically reduce the CFG to a linear sequence of macro-blocks, some of which are just basic blocks (simple macro-blocks), while the others are the content of the SISO region (complex macro-blocks). This allow us to compute the average number of spills/fills in every execution path of the function as the sum the average number of spills/fills in each macro-block. The macro-blocks identification is performed by traversing the dominator tree of the function. For each node we visit, we count the number of children in the tree. If a node has at most one child, it is marked as single macro-block and the next node to visit is its

own child. If the node has more than one child, it is the entry of an SISO region, thus, one of its children is the exit node of the region. To find the exit node we simply check which child is the immediate post-dominator of the entry node. Once the region is identified the dominator tree traversal moves on from the exit node of discovered region. For simple macro-blocks we just count the number of spill/fill operations. For complex macro-blocks we compute the average number on every path in the SISO region excluding the exit node. This is a sound estimate for the effective average number of spills/fills as all the paths in the region are taken with the same probability by construction. In case the block cipher implementation is not fully unrolled, the number of iterations for each loop must be taken into account. The assumption dealing with block ciphers is that every loop has a constant trip count, as it is one of the crucial security parameters. In this case, we simply identify the loop bodies in the linear sequence of macro-blocks and we multiply the average number of spills/fills in each loop body by its trip count before adding them up. The approach employed to compute the number of spills/fills is also extended to the computation of the average number of instructions per execution. The IPC is derived from the execution time measurement, considering the 168 MHz working frequency of the Cortex-M4 under exam and the aforementioned instruction count per execution.

To evaluate the memory fingerprints of the examined algorithms, we measured the sizes of the `.text` (code segment) and `.data` (writable statically initialized variables)/`.rodata` (constant data) sections of the produced executables (results are reported in Table II). Both `.text` and `.rodata` sections are mapped to the μC flash memory, while the `.data` segment is allocated in the integrated SRAM.

C. ISO-Standard Cipher-Suite Evaluation

In our performance evaluation, we provide a complete survey of all the general purpose block ciphers standardized by the ISO committee in ISO-18033-3 [26] (general purpose block ciphers), i.e., AES, Camellia, SEED, Triple DES (TDES) (also known as Triple DEA), CAST128, MISTY1, and HIGHT, together with the ones standardized for use as lightweight block ciphers in ISO-29192-2 [27], i.e., PRESENT and Clefia. In addition to the aforementioned block ciphers, we also provide results on XTEA [32], a lightweight block cipher which is known to be very frequently used in software implementations for embedded environments due to its remarkably good performances and reasonable security margin. We note that, to provide a fair comparison, all the mentioned block ciphers are evaluated for a 128-bit secret key: a security level deemed sufficient by NIST to provide security beyond 2020 [33]. As a consequence, all the cipher features which may depend on the key length (e.g., variable number of rounds) are described for the 128-bit key version. All the examined implementations were fully unrolled to maximize performances, as the unprotected code size was not an issue on our target platform. We now provide a per-cipher analysis of the results, highlighting the reasons for the different performances of the ciphers, and finally draw the conclusion.

1) AES: It is substitution-permutation network cipher with a 128-bit state, seen as a 4×4 bytes matrix, which is elaborated through a ten-round computation, where each round is composed of: 1) the bytewise replacement of the state values with the ones resulting from a 256-entry table lookup (SUBBYTES primitive); 2) a byte-wise rotation of the state matrix rows (SHIFTROWS primitive); 3) an `xor`-linear combination of the columns of the state matrix (MIXCOLUMN primitive); and 4) the addition of a value derived from the 128-bit secret key via `xor` (ADDROUNDKEY primitive). All the rounds share the same four-primitives structures save for the last, which lacks the application of the MIXCOLUMN, and the first, which is preceded by an extra ADDROUNDKEY. A well-known technique to obtain significant speedups in software AES implementations is the use of the so-called *T* tables, i.e., larger LUTs that combine the effects of the SUBBYTES, SHIFTROWS, and MIXCOLUMN primitives into a set of 4, 1 kiB wide, LUTs. Note that, the four *T* tables contain the same 256, 4 bytes wide, elements, save for a byte-wise rotation of each one of them. The *T* tables implementation is widely adopted by embedded system tailored secure sockets layer (SSL) libraries such as PolarSSL. In particular, the results in Table I report that the unprotected *T* tables based implementation of AES has a significant performance improvement over the *S*-box one on our platform due to a lack of spill actions. The higher-IPC reported by the unprotected *S*-box implementation of AES, which is to be ascribed to a higher computational-to-memory instruction ratio in the code, is not able to make up, in terms of absolute performances, for the higher number of instructions with respect to the *T* tables implementation. The protected AES implementations still have the *T* tables one as the best choice in terms of throughput with an effective latency per encryption action of 80.33 μs, which is still reasonable given the typical interaction times of μC-based embedded systems. Concerning the code sizes, Table II reports that the protected *S*-box implementation results in a significantly larger protected code than the *T* tables one. This effect can be ascribed to the MEET approach protecting all the `xor` operations in the algorithm, which in the *S*-box implementation are abundant due to the need to compute the MIXCOLUMN primitive. On the other hand, the *T* tables based implementation requires a larger writeable data segment, as the masked *T*-table should be refreshed during the execution. However, the writable data requirements amount to 2 kiB, which is well within the capabilities of the chosen target platform.

2) Camellia: It is an 18 rounds Feistel network cipher, which has been selected for the Japan e-Government approved cipher list, as the only one developed in Japan and has been integrated by the internet engineering task force in the supported ciphers for the transport layer security secure transport protocol. Camellia acts on the 128-bit state processing a half of it at each round through a Feistel function, and adding it to the other half via `xor`. After this computation, the two halves are swapped before processing starting to apply again the round transformation.

The Feistel structure of Camellia sports a slight modification, in terms of two applications of a small bitwise Boolean function involving the entire state and some key material, after rounds 6 and 12. This structural tweak has been added to the structure to provide immunity against mathematical cryptanalysis techniques such as linear and differential cryptanalysis. The computation of the Feistel function of Camellia involves lookups into a 8 → 8-bit *S*-box and bitwise rotations. Camellia yields throughputs comparable to AES-T in

TABLE II
CODE AND DATA SEGMENT SIZES FOR UNPROTECTED AND FULLY PROTECTED BINARIES OF ISO-STANDARD CIPHERS

Reference	Cipher	Block Size	Unprotected .text [kiB]	Unprotected .rodata [kiB]	Unprotected .data [kiB]	Unprotected Total size [kiB]	Fully Protected .text [kiB]	Fully Protected .rodata [kiB]	Fully Protected .data [kiB]	Fully Protected Total size [kiB]
ISO-18033-3 Wide Block [26]	AES-S	128	9.13	0.57	0.00	9.7	105.61	0.32	0.50	105.93
	AES-T	128	4.16	1.35	0.00	5.51	33.78	0.35	2.00	34.13
	Camellia	128	5.50	0.67	0.00	6.17	47.80	0.42	0.50	48.22
	SEED	128	5.95	8.06	0.00	14.01	58.07	4.06	8.00	62.13
ISO-18033-3 Narrow Block [26]	3DES2	64	8.79	2.17	0.00	10.96	107.28	0.17	4.00	107.45
	CAST128	64	3.80	8.05	0.00	11.85	20.73	4.05	8.00	24.78
	MISTY1	64	3.98	5.05	0.00	9.03	37.97	2.55	5.00	40.52
	HIGHT	64	8.25	0.17	0.00	8.42	43.11	0.17	0.00	43.28
ISO-29192-2 Lightweight [27]	Clefia	128	16.08	1.30	0.00	17.38	113.64	0.80	1.00	114.44
	PRESENT	64	30.98	0.05	0.00	31.03	90.02	0.05	0.03	90.07
[31]	XTEA	64	3.09	0.05	0.00	3.14	9.82	0.05	0.00	9.87

its unprotected form, while its performances are lower when protected via MEET approach. Such a result is justified by the need to protect `and` and `or` operations together with `xors` during Camellia execution, thus, effectively raising the number of executed instructions further. This instruction increase is also reflected on the code size of Camellia, which turns out to be 14.02 kiB larger than AES-T as far as code size goes. However, Camellia benefits from the small *S*-box when it comes to writable data requirements, reducing the required figure by three times with respect to AES-T.

3) SEED: It is a 16 rounds Feistel cipher with a 128-bit state, developed by the Korea Information Security Agency (KISA), which has found very widespread use in South Korea up to the point of influencing the diffusion of Web browsers, favoring only the ones supporting it. The structure of the Feistel function of SEED is itself a Feistel cipher with a 64-bit state. This recursive design has been proven to be one of the ways to achieve linear and differential cryptanalysis immunity [34], and is also used by the National Security Agency-designed cipher Skipjack. The inner Feistel function combines the key material with the state via `xor`, after the application of one out of two $8 \rightarrow$ 8-bit *S* boxes and a bitwise permutation. In order to obtain an efficient software implementation the bitwise permutation and *S* boxes are combined together into a set of 4, $8 \rightarrow$ 32-bit *S* boxes, the result of which is simply combined via inclusive `or`. The performances of the unprotected SEED cipher are slightly lower than the other wide-block ISO-standard ciphers, mainly due to a higher-structural complexity of the design, which was inserted to improve the security margin. This effect is reflected both in terms of reduced performances, and increased code size in the protected version of SEED which, however, still fits into our target platform, occupying only 5% of the permanent storage and 3% of the SRAM.

4) Triple DES: It is an extension of the DES cipher devised by NIST to cope with the short (56-bit) key length of DES, which had become vulnerable to brute-force attacks in 1998, without a full cipher redesign. TDES is built as the concatenation of three executions of the DES block cipher, each one acting on the 64-bit state resulting from the computation of the previous one. The DES block cipher is a 16 rounds Feistel cipher, preceded and followed by a bitwise key-independent permutation designed to ease wiring in hardware, which is thus left out of the MEET protected code region. As the initial permutation is exactly the inverse of the final one, during the computation of TDES, the intermediate permutations are skipped altogether for performance reasons. The Feistel function of DES is constituted by two bitwise permutations and a key material addition via `xor` together with a series of lookups into a set of 8, $6 \rightarrow$ 4-bit *S* boxes. Similar to SEED, the bitwise permutations and *S*-box lookups are implemented by employing larger *S* boxes, and recombining the result via inclusive `or`. TDES is demanding both performance-wise and code-size-wise, as it is to be expected from the very high number of rounds of this cipher (i.e., 48). These issues, will also influence the performances and code size of the protected implementation of TDES, are the result of TDES being a stopgap solution to the cracking of the single DES cipher. To this end, TDES should be chosen to be implemented in an embedded environment only when backward compatibility is desired. However, the MEET protected TDES implementation is able to achieve a block encryption in around 0.25 ms, which is still reasonable in case TDES is used in an interactive environment.

5) CAST128: It is also known as CAST5 is a 16 rounds Feistel cipher, which has been approved for Canadian government and, due to its early royalty free availability has been chosen as the default block cipher by the GNU Privacy Guard e-mail encryption software suite. CAST128 acts on a 64-bit state, and is designed for fast software execution on 32-bit platforms. The Feistel function relies on a combination of `xor`, addition and subtraction modulo 2^{32} both to add the key material and provide diffusion and nonlinearity, in addition to a layer of four $8 \rightarrow$ 32-bit *S* boxes. The positions where the operations are employed in the Feistel function are rotated at each round, thus, effectively yielding three different functions with substantially the same computational complexity. While the design choice of CAST128 requires the MEET approach to protect `add` and `sub` operations in addition to the `xor`, the compact code size obtained by employing arithmetic operations to yield fast diffusion still pays off. The protected implementation of CAST128 occupies less than 2% of the nonvolatile memory of the target device, while providing throughputs comparable to AES-T, despite its narrow block design.

6) MISTY1: It is an 8-round Feistel cipher, which, thanks to its low hardware requirements has been adopted as the building block for A5/3-KASUMI, the current standard cipher to provide confidentiality in 3G mobile data exchange. MISTY1

acts on a 64-bit state, applying once every 2 Feistel rounds a nonlinear function of the key bits to the entire state following the same design decision taken in Camellia. The Feistel function in MISTY1 is itself a 3-round Feistel cipher employing two asymmetric *S* boxes, a $7 \rightarrow 7$-bit and a $9 \rightarrow 9$-bit together with combining the key with via `xor`. MISTY1 effectively provides a reasonable throughput, which is comparable with wide block alternatives such as SEED, and a low-encryption latency which makes it a good choice in scenarios where only a few blocks should be encrypted. The code size of the MEET protected MISTY1 implementation is among the lowest in the examined cipher suite, while the writable memory requirement are slightly higher due to the 9-bit *S*-box which cannot be fit efficiently in software implementations due to lacking byte alignment.

7) HIGHT: It is a compact block cipher designed by KISA, exploiting a reduced amount of hardware computational and storage resources, and thus, aimed at low-area hardware implementations. The design of HIGHT is based on 32 rounds acting on the 64-bit state only through rotations, byte-wise `xor` and additions, in order to allow an efficient implementation of the cipher in hardware. At each round, all the bytes in position $2n$, $1 < n < 4$ of the state are recombined with a rotated version of themselves via `xor` and, depending on their position, are added to the key and their corresponding $2n-1$ placed byte via either `xor` or addition modulo 2^8. Despite the very high number of rounds (32), and the requirement to protect both `add` and `xor` operations HIGHT still retains MEET protected performances close to the one of comparable alternatives such as MISTY1. The code size of the protected implementation of HIGHT is also similar to the ones of other narrow block alternatives and smaller than the most wide-block ciphers, mainly due to the simple structure of its round, which compensates for the higher number of unrolled rounds.

8) PRESENT: It is a 32 rounds, 64-bit state, substitution-permutation network cipher, which was explicitly designed as a candidate substitute for stream ciphers in very low-power hardware implementations (e.g., radio frequency identification (RFID) devices), and has been standardized by the ISO/IEC in [26]. The present round structure is remarkably simple as it applies single a $4 \rightarrow 4$-bit *S*-box to the whole state, followed by a bitwise permutation and a key addition via `xor`. The software implementation of PRESENT on our target platform is the one provided by the authors as optimized for memory size, and thus, provides a rather limited throughput. However, given the typical communication bandwidth of extremely constrained devices such as RFIDs, the 36.35 kiBs of throughput provided by the implementation are usually more than enough to cope with the requirements. A notable aspect of the memory fingerprint of PRESENT is the fact that its writable data section is among the smallest at all, taking up less than 0.1% of the device SRAM thanks to its extremely small *S*-box. The code size of the fully unrolled cipher takes up 8% of the device memory, and can be further reduced significantly, avoiding the unrolling of the 32 rounds.

9) Clefia: It is a four branch Feistel cipher with 18 rounds acting on a 128-bit state designed by Sony for use in Digital Rights Management systems and is among the Japanese government recommended cipher from the CRYPTREC initiative [35]. This design strategy relies on two quarters of the state being processed by two Feistel functions in parallel, and being combined with the other half of the state after the processing. This structure allows Clefia to have a more compact Feistel function, at the cost of adding additional rounds to make up for the reduced diffusion capability [36]. To provide a strengthened structure against algebraic attacks, Clefia alternates two different Feistel functions, both realized as a $8 \rightarrow 8$-bit *S*-box substitution layer, with a different *S*-box per Feistel function, a column-matrix multiplication similar to the MIXCOLUMN. The rather complex structure of Clefia, which is geared toward compact hardware implementation results in a relatively low software throughput for the MEET protected implementation, even though it compares favorably to the other lightweight alternative, PRESENT. However, due to the $8 \rightarrow 8$-bit *S* boxes in Clefia, its writable memory requirements are in line with common, nonlightweight ciphers.

10) XTEA: It is an extremely compact, 64 rounds Feistel cipher, widely known for its low requirements in both software and hardware implementations [31]. Despite not being standardized by international committees, its public domain availability and great simplicity have fostered its success, and its implementations are very common in embedded systems, among which one is available within the Linux Kernel cryptographic facilities. The XTEA Feistel function is constituted of one 32-bit addition, two rotations, and an `xor` addition with key material; the output of the function is combined with the other half of the state via a 32-bit addition. Thanks to its extremely compact form, XTEA achieves a significant throughput in the MEET protected implementation, yielding more than $4\times$ the performances of AES, together with a very compact code size and the lack for a writable data section. These results can be ascribed to both high simplicity of the cipher, and its working set fitting completely into the registers.

11) Summary: Summing up the obtained results in terms of performances, all the wide block ISO standard candidates provide throughputs between 100 and 200 kiB/s when protected with MEET. Such an encryption throughput allows to encipher the whole content of the platform volatile memory in 1–2.5 s, which represents a practical usability validation for the approach, regardless of which of the three candidates are selected due to the platform requirements or governmental regulations. The code sizes of the protected implementations, reported in Table II, are in the 10–113 kiB range, thus, providing viable solutions for the average microcontrollers currently available on the market, where the flash memory employed to store the code segment is in the hundreds of kiB range. Among the narrow block candidates, CAST128 represents a remarkably good solution in terms of both high protected throughput, and reduced code size even when fully unrolled. The performance results provided by Clefia and PRESENT are not up to par with the general purpose ciphers: although this may sound counter-intuitive, given their lightweight classification, it is in fact to be quite expected. This is due to the lightweight classification referring to the low amount of physical resources required to implement them in hardware, which is obtained trading off performances, as explicitly stated in the PRESENT design document [37]. Still, the throughputs provided by both Clefia and PRESENT are more than sufficient in terms of providing confidentiality in communication to very low-resource devices such as RFIDs, which typically have communication bandwidths of some kiB/s. It is worth mentioning that XTEA, despite not being officially

standardized, preserves its performance advantage once protected with the MEET approach: it is thus a worthy alternative for constrained systems whenever ISO/IEC standard compliance is not an explicit requirement. One interesting thing to be noted is that some unprotected ciphers are characterized by a rather low IPC (e.g., 0.31 for AES-T), despite their throughput being good on the target platform. This effect is a consequence of the three cycles latency required for memory access and the high amount of memory lookups which characterizes their implementations. As a consequence, applying the MEET protection may lead to higher IPCs (0.56 for AES-T) on the protected implementations, due to the increase in the percentage of computational instructions. Such a change in the instruction assortment of the implementation may provide a lower overall time penalty than the one predicted simply from the increased instruction count, due to a lower number of pipeline stalls.

V. Related Work

A. Automated SCA Countermeasures

Power management techniques available in general purpose processors are employed in [38] to implement a hiding countermeasure. They rely on external OS support for dynamic voltage/frequency scaling, which may not be available in all platforms. An automated approach to "hiding" countermeasures for software cipher implementations is proposed in [20], through the analysis of possible instruction schedules of the cipher code. These techniques are orthogonal to the one presented in this paper and can be implemented with our approach. Moss *et al.* [39] proposed a first attempt at automating the process of inserting a first-order masking scheme in the code of AES using an *ad hoc* translator. Their scheme relies primarily on type inference, a kind of static analysis which is strongly dependent on the source language. The solution in [13] proposes a dynamic morphing approach. For dynamic morphing, the overhead for a single code morphing action is significant with respect to an unprotected cipher execution. It is worth noting that the experiments in [13] protect only the computational instructions (`xor`), not the table lookups, nor eventual spill and fill actions. The overhead introduced by the MEET approach for protecting only the computational instructions is comparable with that of dynamic morphing. Finally, MEET does not require code execution from a writable memory region, thus, allowing it to run on platforms where nonwritable code segments are physically enforced, such as microcontrollers.

B. Hardware-Inspired and Protocol Level Countermeasures

Hardware countermeasures based on the principle of computing both a value and its Boolean complement simultaneously through duplicating the circuits, provide an effective protection by equalizing the power consumption of different computations. Such protected circuits can be emulated in software [40], obtaining a countermeasure that is independent of the specific cipher and implementation, albeit at a very significant computational cost. All-or-nothing transforms are used in [41] to strengthen ciphers against differential power attacks. While this technique can be automatically applied at a protocol level, it has the downside of increasing the size of the messages to be encrypted (and transmitted) by at least a factor of two. However, no performance results are provided on the application of this approach to a real-world platform.

C. Automated Analysis of SCA Vulnerability

The first work tackling the problem of automatically protecting software implementations of cryptographic algorithm from power analysis attacks is presented by Bayrak *et al.* [12]. In [11], a fully automated analysis to identify vulnerable instructions is proposed, which does not need any profiling information about the power consumption of the target device while it is running the implementation of a cryptographic primitive, and is not affected by completeness and/or accuracy problems bound to the collection of the power measurements. In [42], an attempt is made to provide a formal assessment of the security of applied countermeasures via formal verification tools, focusing on masking techniques. Cobb *et al.* [43] introduced a methodology to assess the resistance of a cipher implementation to SCAs comparing the leakage obtained on measurements with a prediction extracted from every intermediate value of the algorithm.

VI. Conclusion

We introduced the MEET approach, a cost effective method for securing cryptographic embedded software against passive SCAs. MEET has no need of specialized hardware resources, and significantly reduces the computation time with respect to the state-of-the-art approaches, forfeiting also the need of the code segment to be writable, which was required in [13]. MEET also solves the issue of safe recomputation of block cipher LUTs, a critical and open issue in protecting software implementations [18].

References

[1] S. Chari, C. Jutla, J. R. Rao, and P. Rohatgi, "A cautionary note regarding evaluation of AES candidates on smart-cards," in *Proc. 2nd AES Candidate Conf. (AES2)*, Rome, Italy, 1999, pp. 1–15. [Online]. Available: http://csrc.nist.gov/archive/aes/round1/conf2/aes2conf.htm

[2] P. C. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *J. Cryptrograph. Eng.*, vol. 1, no. 1, pp. 5–27, 2011.

[3] M. Rostami, F. Koushanfar, J. Rajendran, and R. Karri, "Hardware security: Threat models and metrics," in *Proc. ICCAD*, San Jose, CA, USA, 2013, pp. 819–823.

[4] J. Heyszl, S. Mangard, B. Heinz, F. Stumpf, and G. Sigl, "Localized electromagnetic analysis of cryptographic implementations," in *Topics in Cryptology—CT-RSA* (LNCS 7178). Berlin, Germany: Springer, 2012, pp. 231–244.

[5] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Comput. Netw.*, vol. 48, no. 5, pp. 701–716, 2005.

[6] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks—Revealing the Secrets of Smart Cards*. New York, NY, USA: Springer, 2007.

[7] E. Peeters, *Advanced DPA Theory and Practice—Towards the Security Limits of Secure Embedded Circuits*. New York, NY, USA: Springer, 2013.

[8] J. Irwin, D. Page, and N. P. Smart, "Instruction stream mutation for non-deterministic processors," in *Proc. ASAP*, San Jose, CA, USA, 2002, pp. 286–295.

[9] J. A. Ambrose, R. G. Ragel, and S. Parameswaran, "A smart random code injection to mask power analysis based side channel attacks," in *Proc. CODES+ISSS*, Salzburg, Austria, 2007, pp. 51–56.

[10] I. Kizhvatov, "Side channel analysis of AVR XMEGA crypto engine," in *Proc. WESS*, Grenoble, France, 2009, Art. ID 8.

[11] G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi, "Compiler-based side channel vulnerability analysis and optimized countermeasures application," in *Proc. DAC*, Austin, TX, USA, 2013, pp. 1–6.

[12] A. G. Bayrak, F. Regazzoni, P. Brisk, F.-X. Standaert, and P. Ienne, "A first step towards automatic application of power analysis countermeasures," in *Proc. DAC*, New York, NY, USA, 2011, pp. 230–235.

[13] G. Agosta, A. Barenghi, and G. Pelosi, "A code morphing methodology to automate power analysis countermeasures," in *Proc. DAC*, San Francisco, CA, USA, 2012, pp. 77–82.

[14] J. Cleemput, B. Coppens, and B. De Sutter, "Compiler mitigations for time attacks on modern x86 processors," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–20, 2012.

[15] C. Lattner *et al.*, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. CGO*, San Jose, CA, USA, 2004, pp. 75–88.

[16] R. Muth and S. A. Watterson, "Code specialization based on value profiles," in *Static Analysis (SAS)* (LNCS 1824). Berlin, Germany: Springer, 2000, pp. 340–359.

[17] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "A multiple equivalent execution trace approach to secure cryptographic embedded software," in *Proc. DAC*, San Francisco, CA, USA, 2014, pp. 1–6.

[18] M. Tunstall, C. Whitnall, and E. Oswald, "Masking tables—An underestimated security risk," in *Fast Software Encryption* (LNCS 4593). Berlin, Germany: Springer, 2013, pp. 290–308.

[19] Y. Ishai, A. Sahai, and D. Wagner, "Private circuits: Securing hardware against probing attacks," in *Advances in Cryptology—CRYPTO* (LNCS 2729). Berlin, Germany: Springer, 2003, pp. 463–481.

[20] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "Enhancing passive side-channel attack resilience through schedulability analysis of data-dependency graphs," in *Network and System Security* (LNCS 7873). J. Lopez, X. Huang, and R. Sandhu, Eds. Berlin, Germany: Springer, 2013, pp. 692–698.

[21] J.-S. Coron, "Higher order masking of look-up tables," in *Advances in Cryptology—EUROCRYPT* (LNCS 8441). Berlin, Germany: Springer, 2014, pp. 441–458.

[22] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F.-X. Standaert, "On the cost of lazy engineering for masked software implementations," *Cryptology ePrint Archive*, Rep. 2014/413, 2014.

[23] P. Bonzini and L. Pozzi, "Polynomial-time subgraph enumeration for automated instruction set extension," in *Proc. DATE*, Nice, France, 2007, pp. 1331–1336.

[24] S. Guilley *et al.*. (2013). *MARSHAL+: Mechanism Against Reverse Engineering for Secure Hardware, PHISIC*. [Online]. Available: http://marshalplus.org

[25] T. Eisenbarth, C. Paar, and B. Weghenkel, "Building a side channel based disassembler," in *Transactions on Computational Science* (LNCS 6340). Berlin, Germany: Springer, 2010, pp. 78–99.

[26] *ISO/IEC 18033-3: Information Technology. Security Techniques—Encryption Algorithms—Part 3: Block Ciphers*, ISO/IEC JTC Standard 1/SC 27, 2010.

[27] *ISO/IEC 29192-2: Information Technology. Security Techniques—Lightweight Cryptography—Part 2: Block Ciphers*, ISO/IEC JTC Standard 1/SC 27, 2012.

[28] STMicroelectronics. (2014). *Discovery Kit for STM32F407/417 Lines—With STM32F407VG MCU*. [Online]. Available: http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/LN1199/PF252419

[29] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM side—Channel(s)," in *Cryptographic Hardware and Embedded Systems—CHES* (LNCS 2523). B. S. Kaliski, Jr., Ç. K. Koç, and C. Paar, Eds. Berlin, Germany: Springer, 2002, pp. 29–45.

[30] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, "A testing methodology for side-channel resistance validation," in *Proc. NIST Non-Invas. Attack Test. Workshop (NIAT)*, Nara, Japan, 2011, pp. 1–15. [Online]. Available: http://csrc.nist.gov/news_events/non-invasive-attack-testing-workshop/

[31] J.-P. Kaps, "Chai-tea, cryptographic hardware implementations of xTEA," in *Progress in Cryptology—INDOCRYPT* (LNCS 5365). D. R. Chowdhury, V. Rijmen, and A. Das, Eds. Berlin, Germany: Springer, 2008, pp. 363–375.

[32] R. M. Needham and D. J. Wheeler, "TEA extensions," Comput. Lab., Univ. Cambridge, Cambridge, U.K., Tech. Rep., 1997.

[33] National Institute of Standards and Technology. (2012). *Recommendation for Key Management, Special Publication 800-57 Part 1 Rev. 3*. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/key_management.html

[34] K. Nyberg and L. R. Knudsen, "Provable security against a differential attack," *J. Cryptol.*, vol. 8, no. 1, pp. 27–37, 1995.

[35] Cryptography Research and Evaluation Committees. (2014). *Cryptrec Report Archive*. [Online]. Available: http://www.cryptrec.go.jp/english/report.html

[36] T. Shirai, K. Shibutani, T. Akishita, S. Moriai, and T. Iwata, "The 128-bit blockcipher CLEFIA (extended abstract)," in *Fast Software Encryption* (LNCS 4593). A. Biryukov, Ed. Berlin, Germany: Springer, 2007, pp. 181–195.

[37] A. Bogdanov *et al.*, "PRESENT: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems—CHES* (LNCS 4727). P. Paillier and I. Verbauwhede, Eds., Berlin, Germany: Springer, 2007, pp. 450–466.

[38] A. Krieg, J. Grinschgl, C. Steger, R. Weiss, and J. Haid, "A side channel attack countermeasure using system-on-chip power profile scrambling," in *Proc. IOLTS*, Athens, Greece, 2011, pp. 222–227.

[39] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler assisted masking," in *Cryptographic Hardware and Embedded Systems—CHES* (LNCS 7428). Berlin, Germany: Springer, 2012, pp. 58–75.

[40] Z. Chen, A. Sinha, and P. Schaumont, "Using virtual secure circuit to protect embedded software from side-channel attacks," *IEEE Trans. Comput.*, vol. 62, no. 1, pp. 124–136, Jan. 2013.

[41] R. McEvoy, M. Tunstall, C. Whelan, C. Murphy, and W. Marnane, "All-or-nothing transforms as a countermeasure to differential side-channel analysis," *Int. J. Inf. Security*, vol. 13, no. 3, pp. 291–304, 2014.

[42] H. Eldib, C. Wang, and P. Schaumont, "SMT-based verification of software countermeasures against side-channel attacks," in *Tools and Algorithms for the Construction and Analysis of Systems* (LNCS 8413). Berlin, Germany: Springer, 2014, pp. 62–77.

[43] W. E. Cobb, R. O. Baldwin, and E. D. Laspe, "Leakage mapping: A systematic methodology for assessing the side-channel information leakage of cryptographic implementations," *ACM Trans. Inf. Syst. Security*, vol. 16, no. 1, 2013, Art. ID 2.