Automated Instantiation of Side-Channel Attacks Countermeasures for Software Cipher Implementations*

Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Piazza Leonardo da Vinci, 32 – 20133 Milano, Italy
name.surname@polimi.it

July 25, 2019

Abstract

Side Channel Attacks (SCA) have proven to be a practical threat to the security of embedded systems, exploiting the information leakage coming from unintended channels concerning an implementation of a cryptographic primitive. Given the large variety of embedded platforms, and the ubiquity of the need for secure cryptographic implementations, a systematic and automated approach to deploy SCA countermeasures at design time is strongly needed. In this paper, we provide an overview of recent compiler-based techniques to protect software implementations against SCA, making them amenable to automated application in the development of secure-by-design systems.

Keywords – Design for security, Side channel attacks, compiler technology

1 Introduction

The widespread use of ultra-low power pervasive computing devices, both as means to drive cyber-physical systems and to provide environmental and health sensing, has lead to a significant increase in the amount of sensitive and security-critical data managed by them. Practical examples of application domains include, financial transactions via smart-cards, vehicle-to-vehicle communications, industrial sensor networks and process control, and smart medical devices. Cryptographic primitives are the foundational building-blocks to provide security and privacy assurances in complex computational and communication systems. Indeed, a significant amount of commercially available

*This is a preprint of the position paper with the same title appeared at ACM Computing Frontiers 2016, http://dx.doi.org/10.1145/2903150.2911707

embedded devices is either endowed with a hardware cryptographic primitive accelerator, or an optimized software library to provide data and communication security and privacy.

However, in such a scenario, an additional threat model with respect to the usual ones, opens up. An attacker may gain physical access to the target device and can effectively exploit such access as a further advantage. Preventing the adverse effects of such a threat model mandates a combined engineering effort in choosing cryptographic primitives which are sound from a theoretical standpoint and carefully implemented so that the large class of the so-called *implementation attacks* are warded off. The choice of sound primitives can be effectively performed picking them among the well scrutinized ones, which have been recognized as standards by international and national entities such as the ISO/IEC committee or the US National Institute of Standards and Technology (NIST), and choosing appropriate key lengths. By contrast, warding off implementation attacks is still a challenging issue, and thus have been a stimulating research topic.

The largest class of implementation attacks is represented by the so-called *side-channel attacks* (SCAs), where the attacker exploits the information leakage happening on an unintended channel, typically an environmental parameter of the computation which is dependent on the computed data. Instances of such side-channels include energy consumption, execution timing or electro-magnetic (EM) emanations: all these environmental parameters provide enough information to infer the value of secret data intended to be stored within the device in an otherwise un-accessible way.

In this work, the focus is on energy consumption based SCAs against block cipher implementations, since open literature reports results of the successful breach of many systems employing them which range from electronic tickets [18], intellectual property protection schemes on large scale reconfigurable devices [22] to software implementations running on high end System on Chips (SoCs) endowed with a full fledged operating system [12].

Designing efficient and effective countermeasures against side-channel attacks is a topic which has received warm attention by the research community. Typically, countermeasures against the aforementioned threats involve modifying the cipher at either the algorithmic or the implementation level [4–6, 8], or changing the underlying hardware architecture so to suppress the side-channel leakage.

In particular, while several works tackled the problem of providing security oriented solutions for hardware designs [19, 23, 27], it is worth noting that a significant number of embedded systems are built on top of general purpose platforms, and thus rely on software-based encryption primitives. Software solutions provide a greater design flexibility, a feature which has been acknowledged by the standardized cryptographic protocols allowing a choice in the algorithms to be employed. Software-based security layers are also employed as a fall-back solution in case the hardware based ones are compromised, an increasing trend given the technological progress of integrated circuits debugging and testing tools [14].

A significant number of countermeasure strategies for software implementations of cryptographic primitives were proposed as tailored modifications to a given cryptographic primitive so that its computation would not directly depend on the input data and the secret parameters only. An alternate approach involves raising the technical

difficulty of gathering proper measurements of the side-channel, effectively hiding the information sought by the attacker either inserting random delays in the computation, or performing useless operations with the sole purpose of providing a smokescreen for the useful ones [24].

The aforementioned strategies, albeit effective in foiling the efforts of the attacker, were typically implemented tailoring them for a specific hardware-software stack, with a significant number of them being encoded in assembly language. As a consequence, tackling the challenge of providing side-channel security to the wide variety of devices and architectures of modern embedded systems was particularly time consuming in terms of re-engineering effort. This represented a significant hindrance to the prompt adoption of such securization techniques on novel systems, due to the significant development time requirements.

In order to overcome the said hindrance, a need for the automated design time detection of side-channel vulnerabilities, and similarly the automated application of countermeasures had risen. The first works aimed at highlighting the extent of the information leakage on the energy consumption side-channel either employing direct measurements on the target platform [13], or performing a static analysis of the code at translation time [5]. Static analysis techniques have also proven successful in detecting faulty countermeasures for which `C` implementations are provided. In particular, in [17] the authors describe a Satisfiability Modulo Theorem solver to either state the correctness of a `C` implementation, or provide a counterexample in the form of a viable side-channel attack strategy.

Complementing the aforementioned solutions, which aim at discovering automatically the extent of the side-channel vulnerability, another line of research explored the automatic application of countermeasures to vulnerable portions of existing implementations. In [2, 5] the automatic application of countermeasures adding random values to the vulnerable portion of the computation of a software implemented block ciphers was realized as a set of passes in the LLVM compiler toolchain. Similarly, in [9] the automated application of hiding countermeasures to software implementations was investigated. In particular, such a study examines the possible schedules of block cipher instructions (at the level of LLVM intermediate representation), to execute the cipher instructions in a different (legitimate) sequence at each run of the primitive and minimize the computational overhead involved.

In addition to the automated application of existing countermeasures, in [4, 6] the authors proposed a substantially different, and automatically applicable, approach which changes dynamically the way a computation is performed, thus hindering the attempts at building a model of how the sensitive information is leaked on the side-channel altogether. Finally, in [7], the authors propose to automatically insert plausible computations with fake keys to act as a bait for an attacker exploiting information leakage on the side-channel. Such an approach results in the attacker retrieving both the correct key and the fake ones with the same confidence, thus forcing him to attempt to breach the system with a possibly invalid key. Such an action can be detected, providing a way to spot an SCA attempt and actively react to it, e.g., by deleting the secret key from the device or rendering it inoperable.

The rest of the paper briefly introduces the framework of an energy based SCA, and describes the countermeasure strategies proposed. A review of the aforementioned

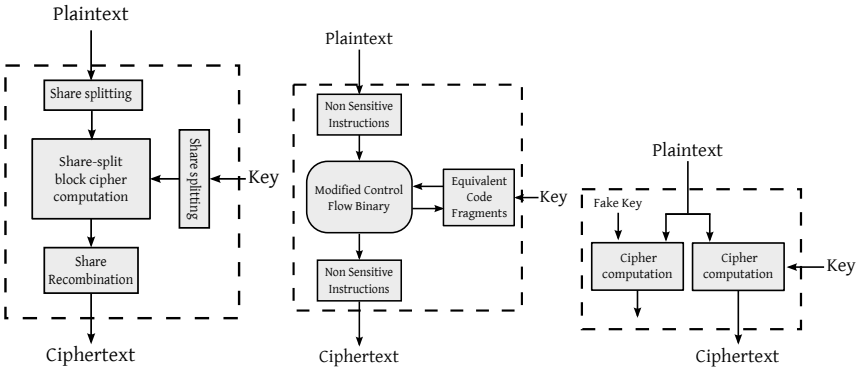
recent approaches to automated countermeasure application at design time is then provided, highlighting their advantages and disadvantages.

2 SCAs and Countermeasures

The typical workflow of an energy consumption based SCA recovers the value of the secret parameter of a cipher (i.e., the secret key) one portion at a time. This is possible since, during a cryptographic computation, the algorithm combines the secret key bits with other input/intermediate values involving a limited quantity of the former at a time. The first step of the attack consists in measuring the energy consumption of the target device, while computing the cryptographic primitive on a large set of different (known) input messages. The measurements are performed either through inserting a shunt resistor on the supply line of the device or measuring the EM emissions radiated by it. Subsequently, an intermediate operation employing a small portion of the secret key is selected, and hypotheses on its results are made for each of the inputs fed to the circuit, and for each value the small key portion involved may take. Such hypotheses on the results are used to derive a sequence of predictions of the energy consumption for each possible value taken by the key portion. Finally, the predicted values are compared with the actual measured ones through the use of statistical means (e.g., linear correlation index or difference-of-means test) to find out which prediction fits best, thus inferring the correct value of the key portion involved. Countermeasures aimed at protecting cipher implementations are traditionally split in two large categories *hiding* and *masking* [24].

Hiding For software implementations, these strategies hinder the matching between the actual power measurements and the hypothesized consumption for each key-portion guess through rescheduling some instructions, permuting the sequence of accesses to lookup tables, or inserting random delays built out of dummy operations [16, 24, 26]. The effectiveness of the hiding countermeasures relies on the fact that the statistical test employed to determine the correct key guess is computed considering the side-channel measurements timewise. As a consequence, randomizing the point in time where a given instruction is computed causes samples from the side-channel measurement of an operation different from the targeted one to be misinterpreted as useful ones, effectively adding noise to the accuracy of the statistical test employed. The security margin provided by hiding countermeasures is typically quantified in terms of the increase in the amount of side-channel measurements caused by the added noise: such an increase is shown to be scaling as the square root of the number of unrelated operations performed in the same time instant as the one under attack.

Masking This countermeasure invalidates the correlation between the values employed to predict the power consumption and the actual values processed by the device [20, 24]. The principle is to add one or more random values (a.k.a., *masks*) to every sensitive intermediate variable occurring during the computation. Sensitive variables are the ones storing a value influenced by a portion of the cipher key (directly or indirectly). In a masked implementation, each sensitive intermediate value is repre-



(a) Masking countermeasure (b) Morphing countermeasure (c) Chaff countermeasure

Figure 1: High level overview of the functioning of the three side-channel attack countermeasure strategies.

sented as split in a number of *shares* (containing both the randomized sensitive value and the masks employed), which are then separately processed. For example, the original sensitive value is xor-ed with $s-1$ random numbers. In this way the original value is put into a one-to-one correspondence with the set of s values (i.e., shares) needed to reconstruct it. The target algorithm is modified to perform the entire computation on the set of share-split values recombining them only at the end of the computation, as depicted in Figure 1a. This technique effectively hinders the attacker from formulating a correct power consumption model. Indeed, the instantaneous power consumption is independent from the original (non-masked) value, as unpredictable random values are newly generated at each execution of the algorithm. Typically, masking techniques are categorized by the number of masks, d, employed for each sensitive value, which is known as the *order* of the masking. A d-th-order masking can always be theoretically broken by a $(d+1)$-th-order attack. Such an attack exploits the combination of $d+1$ measurements of the computations of different shares during the same cipher execution, to build a mask-independent power consumption model [24]. This in turn provides a tight upper bound to the capabilities of an attacker able to breach any d-th order masking scheme. In principle, the lower bound on the number of measurements needed to breach a generic masking scheme amounts to a single measurement, depending on the specific operation to be protected and the specific structure of protection scheme. Besides the theoretical upper limit imposed by the number of shares (i.e., $d+1$), open literature also provides instances of masking schemes where a lower bound tighter than 1 on the number of measurements is proven formally.

Provably Secure Countermeasures The first masking scheme providing a formally proven lower bound on the attacker effort was described by Ishai et al. [20], and is commonly referred to as *ISW masking* from the initials of the authors. A d-th order SCA resistant algorithm employing the ISW masking scheme provably requires the attacker to be able to perform a *d+1* measurements in order to breach the protection.

Table 1: Complexity of bitwise masked operations as a function of the masking order d and lookup table size l

Op.s	Complexity	Ref.
`xor`	$3(d{+}1)$ `xor`	[20]
`not`	1 `not`	[20]
`and`	$3d(d{+}1)$ `xor` $+2d^2{+}3d{+}1$ `and`	[20]
`or`	$3d(d{+}1)$ `xor` $+$ $+2d^2{+}3d{+}1$ `and` $+3$ `not`	$a \vee b{=}\neg((\neg a) \wedge (\neg b))$
table lookup	$ld(3d+1) + 2d$ `xor` $+ld(d+1)$ `store` $+$ $+ld(d+1)$ `load`	[15]

At the beginning of the sensitive computation, the ISW masking scheme performs the splitting of the input values into s shares each. In particular, each input value is added, via `xor`, to $s{-}1$ random values to obtain the first share, while the random values themselves are considered as the remaining ones. To recombine all the shares of the result at the end of the computation, the ISW masking scheme simply adds all s of them together via `xor`. In order to perform the computation over share-split inputs the ISW masking provides a constructive method to transform the unprotected algorithm into a protected version, modeling it as a Boolean circuit. The method provides a strategy to rewrite every single bit Boolean `and` and `not` operations so that they are able to compute a result split in s shares starting from similarly s shared inputs. Although having a method for obtaining protected `and` and `not` operations is already sufficient in itself to implement any Boolean circuit (and thus obtain a protected form of any algorithm), it is possible to implement the protected `xor` operation in the ISW masking framework in a more efficient way than its simple Boolean expansion in a combination of `and`s and `not`s, exploiting the fact that the share-splitting is indeed performed via exclusive-`or` addition of random values. The ISW masking scheme provides security up to a d-th order attack, employing $s{=}2d{+}1$ shares in its first formulation presented in [20], although the authors note that it is possible to reduce the number of shares down to $s{=}d{+}1$, while maintaining the same degree of protection. The threat model assumes an adversary able to acquire at most d simultaneous bit-level values during per clock cycle of the computation [20]. The scheme is proven to provide the indistinguishability of the d values obtained by the attacker from d randomly extracted values, thus providing perfect security of the computation against probing. Table 1 shows the computational costs to protect all common Boolean operations with the ISW masking, employing the modified scheme with $s{=}d{+}1$ shares. Despite the possibility of rewriting any algorithm as a sequence of Boolean operations on its inputs, it is commonplace, for performance reasons, to implement computationally intensive functions as a lookup table where the inputs are employed to index the entry containing the result of the evaluation of the function on them. The only provably secure scheme providing a constructive framework to perform secure table lookups on share split values for any number of shares s is described in [15]. Such a scheme relies on share-splitting the entire tables computing s of them before each load operation is performed, adding fresh

randomness to their encoding. As a consequence, the cost of the protection scheme is quite significant, as reported in Table 1. In particular, the author of [15] highlights that such an approach is profitable only in the case where the computation of the tabulated function requires a high number of nonlinear Boolean operations (i.e., `and`s and `or`s).

3 Automated approaches

This section discusses two recent approaches to automated countermeasure application at design time, as well as a technique to assess the vulnerability of a software implementation to SCAs, highlighting their advantages and disadvantages. All these techniques are implemented by means of specialized compiler passes and can be automatically deployed with minimal intervention of the software developer, who does not need any security background. Thus, the possibility of combining them is worth investigating.

3.1 Automated Vulnerability Analysis

The goal of a SCA vulnerability assessment is to determine the computational difficulty of inferring the secret value employed as the direct or indirect input of a computation from the observation of a physical quantity associated with the intermediate value generated by that computation. The security of a cipher implementation is only as strong as that of the most vulnerable intermediate value computed by any of its instructions. Following the approach described in [5], an instruction is deemed to be vulnerable if it is computationally feasible to compute a model of the physical characteristic of its behavior (e.g., power consumption or EM-emission) for each possible value of the key bits, that concur to the computation of the output value of the instruction. Computing the aforementioned model is the ground on which energy based SCAs are built, as its predictions are matched against the measured behavior of the considered device.

As discussed in Section 2, this computational difficulty depends directly on the number of secret key bits involved in the operation that produced the intermediate value targeted by the attack. Thus, a value computed using only a single bit of the secret key is easiest to attack, whereas the maximum degree of security is achieved by using all bits of the secret key to compute a value. It is worth noting that values that are not computed using any bit of the secret key are irrelevant from the point of view of SCAs, since they carry no useful information.

In [3], a *security-oriented data flow analysis* (SDFA) is introduced, allowing a precise assessment of the vulnerability of each instruction, carried out on the intermediate language of the LLVM compiler [21]. The SDFA, in essence, performs an analysis of the propagation of the key material through the sequence of instructions of the cipher implementation, computing for each bit of an intermediate value the set of cipher key bits from which its computation depends. It is straightforward to understand that, in the first operations of a cipher, only a few bits of the key are combined to generate the intermediate values, but the diffusion properties of the algorithm (usually deemed desirable in a cipher design) guarantee that after a few rounds, all the key bits will be used to compute each bit of any intermediate value. Thus, an attacker can only exploit the first few rounds of the cipher as the target of an SCA, and the SDFA can precisely

identify how many rounds can be practically attacked, as well as the computational effort needed to carry out the attack. Since SCAs can also be carried out considering the ciphertext and targeting the instructions in the last rounds of the cipher, a *backward* version of the SDFA is also introduced to complete the security assessment of the examined implementation. The overall vulnerability of each instruction of the cipher is, once more, the minimum of those reported by the two analyses.

The proposed technique is particularly attractive because it is independent of the source language used to implement the cipher, and is statically computed, without the need to actually run the cipher. In [2], the SDFA has been applied to a wide range of standard block cipher implementations, including the Advanced Encryption Standard (AES), Camellia [11], Triple DES and DES-X, and Serpent [10]. It is worth noting that the SDFA allows to minimize the set of instructions that must be protected, given a desired level of protection. In [2], a masking countermeasure is applied selectively only to the instructions with a vulnerability greater than the desired level of protection, showing that ciphers that are known to be slower in unprotected implementations can actually outperform faster ciphers when countermeasures are applied, thanks to their stronger security guarantees, which reduce the performance overhead of the countermeasure application.

3.2 Code Morphing Countermeasures

The Code Morphing [4] approach aims at altering the side channel profile of the application code, both in terms of power consumption and radiated electromagnetic emissions, thus making the construction of a model of the side channel impossible. Without such a model, the attacker cannot successfully extract the secret information from the side channel.

A modified compiler based on the LLVM framework [21] provides the means to automatically install the necessary countermeasures against passive SCAs. The compiler front-end recognizes custom attributes [4] or additional language keywords [1,6], used to mark code blocks that need to be protected and arrays of constants accessed through key-dependent values. These markers are propagated through the compiler front-end, and preserved by the compiler optimization passes, so that the protected code regions are never violated (e.g., by compiler optimizations that reorder code).

After code optimization, the compiler pass identifies the sensitive instructions in the protected regions, and replaces each of them with a set of semantically equivalent alternatives. Two methodologies are available to achieve this goal. The first employs a *polymorphic engine* embedded in the application code by the compiler. The polymorphic engine replaces at runtime the sensitive instruction with a randomly selected semantically equivalent alternative. The second, on the other end, employs the *MEET pass*, where the sensitive instruction is removed from the code, and replaced with a selection construct, driven by a value randomly generated at runtime, which selects one of the set of equivalent alternatives to the original instruction.

To produce these semantically equivalent alternatives, the MEET pass (or the polymorphic engine) reads a configuration file that stores, for each sensitive instruction, a list of equivalent code fragments, called a *code tile* [4]. In this configuration file, each sensitive instruction is represented in a normalized format that abstracts from the actual registers and constant operands. Figure 2 shows an example of code tile for an `eor`

```
bic r0,r1,r2
bic r3,r2,r1
orr r0,r0,r3
```

```
bic r5,r0,r4
bic r3,r4,r0
orr r5,r5,r3
```

Figure 2: A semantically equivalent fragment for `eor r2,r0,r1`, both in normalized (top) and denormalized form (bottom)

```
and r0,r1,#0⊕const1
and r2,r1,#0⊕¬const1
and r0,r0,r2
```

```
and r0,r1,#0⊕const1
and r2,r1,#0⊕const1
orr r0,r0,r2
```

Figure 3: Two *tiles* for `and r0,r1,C0`

instruction, showing its denormalization to match an original sensitive instruction `eor r2, r0, r1`. Figure 3 shows two code tiles for an `and` instruction, demonstrating the abstraction of constant values.

Access to lookup tables need separate protection, since equivalent code tiles cannot prevent side channel leakage from the memory bus. The *polymorphic engine* protects lookup table accesses through the application of a random permutation to the allocated array indexes, hiding the access patterns to the substitution table of a symmetric cipher. This access pattern hiding technique has been proposed and detailed in [25, 26].

The MEET pass also protects lookup table accesses through a *share splitting* [24] technique. The share splitting technique splits each value to be stored in memory in multiple shares, which are all random numbers except one, which is the bitwise exclusive or of the original value and all other shares. The shares are combined in the CPU registers, so that the secret value never appears in its unprotected form on the memory bus. Special care must be taken to periodically change the random values used to protect the original one, without leaking information during the refresh operation [6].

3.3 Chaff Countermeasures

The *chaff countermeasure* is a defense strategy against a side-channel attacker who has complete knowledge of the details of a software implementation of a block cipher primitive, and is trying to exfiltrate the secret key through exploiting the information leakage during the decryption of a ciphertext. The attacker is assumed to have no means to access the output of the decryption but can only observe the actions performed by the attacked security system. Practical application scenarios include Intellectual Property (IP) protection for post-deployment firmware updates and keyless entry systems. This countermeasure swarms the attacker with fake-but-plausible side-channel leakages among which the real one is blended. This enables the system designer to detect the attacks as soon as a wrong key is employed to forge system inputs, provided that the

[[IMAGE DISCARDED DUE TO '/tikz/external/mode=list and make']]

Figure 4: Decision tree for selecting the appropriate countermeasure strategy

number of plausible alternative values for the secret key is high enough to have an undetected use of a wrong key with negligible probability. This in turn enables a prompt response to a breach attempt before the attack succeeds, a key feature in domains such as automotive, sensor networks and industrial control. Since the fake leakage is not distinguishable from the real one, the security of the proposed defense strategy is not altered even in case more technically challenging attacks, such as High-Order (HO) analyses or template analyses, are employed to attempt a breach. This is in contrast with typical leakage suppression techniques (e.g., masking and hiding), where the defender attempts to hinder the exploitation of the leaked information raising the required technical effort to lead the attack.

To achieve the chaff property, the device behavior should both report more than one key-dependent behavior as correctly fitting, and make such fitness happen in the same time instants. To achieve this, an execution trace randomization technique similar to Code Morphing is employed so that either the instruction computing the real cipher result, or one of its chaff is executed randomly at each cipher run. Such a randomized scheduling causes the fitness for multiple keys to peak apparently simultaneously, since the side-channel analysis combines a statistically significant amount of measurements from different runs together to compute the fitness of the hypotheses timewise.

The execution trace randomization will thus select, through a random-number-generator-driven (RNG-driven) switch-case like construct, one out of many alternate code fragments, each one of which should contain both the real instruction and the corresponding chaff ones. Particular care should be taken in the scheduling of the instructions of each alternate branch of the switch-case construct. Their schedules should be chosen in such a fashion that each instruction is executed an equal amount of times over the same clock cycle, across different runs. A straightforward approach to building these schedules is to emit $(\#\mathrm{chaff} + 1)!$ alternate code fragments, each of which is made of a permutation of the aforementioned instructions. However, the overhead introduced by such an approach grows very quickly in the number of chaff keys. A viable efficient alternative is to build $\#\mathrm{chaff} + 1$ alternate code fragments obtaining each one of them as a sequence of 1 instruction rotations, starting from an arbitrary schedule. The resulting code fragments set fits the chaff property requirements, while retaining an overhead which grows only linearly with the number of chaff keys.

3.4 Comparative analysis

It is now worth considering the applicability of the three types of countermeasures described above (morphing with the polymorphic engine, morphing with the MEET pass, and chaff) in different use case scenarios, comparing them with a standard masking. First, let us consider the computational overhead imposed by the three countermeasures. All the approaches have been applied to AES128 with tabulated S-boxes. The relative overheads (slowdown) are compared, since different boards (albeit all featuring

ARM-based embedded processors) have been used to obtain the experimental results.

A typical first-order masking, applied to the entire cipher, imposes a major penalty, in the range of 100× [3]. Employing the SDFA to pinpoint the vulnerable instructions and protecting only those reduces the overhead to 42× [3].

Code morphing with the polymorphic engine is inefficient if the code is morphed at every call of the protected primitive – the overhead reported in [4] is 392×, greater than most masking approaches. The overhead could be reduced employing the SDFA, but it would still be in the range of 156×. However, it is possible to trade off some security for performance by performing the morphing action only once every n calls, where n can be chosen based on the impact on the overlap of the confidence intervals with which the true and the best false guess are estimated. In [4], it is shown that the value of n can range between 100 and 3000 without significantly affecting the security (the confidence intervals overlap by more than 79%, and the best false key is always identified with a better confidence than the true key), with an overhead reduced to 4× and 0.2× respectively, even without applying the SDFA. The MEET pass, on the other hand, imposes an overhead of 4.82× to 4.33×, depending on the specific architecture, which can be lowered to 2.14×-2.42×, by employing the SDFA to identify the vulnerable instructions and applying the MEET pass only to the AES rounds that include them. The overhead of the chaff countermeasure depends on the number of fake keys desired, which in turn impact the probability the attacker has of guessing the correct key. A single chaff imposes an overhead of 1.37×, whereas with three chaffs the overhead increases to 2.28×. Comparing the above results, the polymorphic engine still provides the fastest countermeasure by one order of magnitude. However, it does have some limitations in its applicability, due to the fact that the polymorphic engine needs the ability to write portions of memory that are labeled as containing executable code. This is not feasible in many microcontrollers. In those cases, the MEET approach provides a suitable alternative. The chaff countermeasure, on the other hand, covers a different use case scenario, where reactive countermeasures are needed in order to detect the attack and raising an alarm (or performing other response actions, including activating self-destruction procedures). The chaff countermeasure is not suitable for scenarios where there is no reaction expected to an attack, because the attacker would then be able to brute force the correct key by trying and verifying all the possible combinations of the false and true key bytes identified through the attack. Finally, the chaff countermeasure cannot be usefully combined with either code morphing approach, but could be combined with standard masking, although the performance impact (in the range of 200×) would make it acceptable only in slow-response systems. Figure 4 summarizes in a flowchart how a system designer would select the appropriate tool for applying SCA countermeasures to a cipher implementation.

4 Concluding Remarks

This paper provided an overview of techniques for protecting software implementations against SCA that can be applied through the use of a specialized compiler, thus making them amenable to automated application in toolchains for the development of secure-by-design systems. These techniques provide protection to the target implementation from different angles, suitable for application in different use case scenarios.

References

[1] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale. The MEET Approach: Securing Cryptographic Embedded Software Against Side Channel Attacks. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 34(8):1320–1333, Aug 2015.

[2] Giovanni Agosta, Alessandro Barenghi, M. Maggi, and Gerardo Pelosi. Design Space Extension for Secure Implementation of Block Ciphers. *IET Computers & Digital Techniques*, 8(6):256–263, 2014.

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

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

[5] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. Enhancing Passive Side-Channel Attack Resilience through Schedulability Analysis of Data-Dependency Graphs. In *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*, pages 692–698. 2013.

[6] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. A multiple equivalent execution trace approach to secure cryptographic embedded software. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 210:1–210:6. ACM, 2014.

[7] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. Information Leakage chaff: Feeding Red Herrings to Side Channel Attackers. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 33:1–33:6. ACM, 2015.

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

[9] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. Trace-based schedulability analysis to enhance passive side-channel attack resilience of embedded software. *Information Processing Letters*, 115(2):292–297, 2015.

[10] Ross J. Anderson, Eli Biham, and Lars R. Knudsen. The Case for Serpent. In *AES Candidate Conference*, pages 349–354, 2000.

[11] Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakajima, and Toshio Tokita. Specification of Camellia-A 128-Bit Block Cipher, September 2001.

[12] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. DPA, Bitslicing and Masking at 1 GHz. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th Int.'l Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 599–619, 2015.

[13] Alessandro Barenghi, Gerardo Pelosi, and Yannick Teglia. Information Leakage Discovery Techniques to Enhance Secure Chip Design. In Claudio Agostino Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication - 5th IFIP WG 11.2 Int.'l Workshop, WISTP 2011, Heraklion, Crete, Greece, June 1-3, 2011. Proceedings*, volume 6633 of *LNCS*, pages 128–143. Springer, 2011.

[14] Christian Boit, Clemens Helfmeier, and Uwe Kerst. Security Risks Posed by Modern IC Debug & Diagnosis Tools. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*, pages 3–11, 2013.

[15] Jean-Sébastien Coron. Higher Order Masking of Look-Up Tables. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pages 441–458, 2014.

[16] Jean-Sébastien Coron and Ilya Kizhvatov. Analysis and Improvement of the Random Delay Countermeasure of CHES 2009. In *Cryptographic Hardware and Embedded Systems*, pages 95–109, 2010.

[17] Hassan Eldib, Chao Wang, and Patrick Schaumont. Formal Verification of Software Countermeasures against Side-Channel Attacks. *ACM Trans. Softw. Eng. Methodol.*, 24(2):11:1–11:24, 2014.

[18] Flavio D. Garcia, Peter van Rossum, Roel Verdult, and Ronny Wichers Schreur. Wirelessly Pickpocketing a Mifare Classic Card. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 3–15, 2009.

[19] Xu Guo, Junfeng Fan, Patrick Schaumont, and Ingrid Verbauwhede. Programmable and Parallel ECC Coprocessor Architecture: Tradeoffs between Area, Speed and Security. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pages 289–303, 2009.

[20] Yuval Ishai, Amit Sahai, and David Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 463–481, 2003.

[21] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the Int'l Symposium on Code generation and optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. On the Vulnerability of FPGA Bitstream Encryption against Power Analysis Attacks: Extracting Keys from Xilinx Virtex-II FPGAs. In *ACM Conference on Computer and Communications Security*, pages 111–124, 2011.

[23] Sri Hari Krishna Narayanan, Mahmut T. Kandemir, and Richard R. Brooks. Performance Aware Secure Code Partitioning. In *Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007*, pages 1122–1127, 2007.

[24] Eric Peeters. *Advanced DPA Theory and Practice - Towards the Security Limits of Secure Embedded Circuits*. Springer New York, 2013.

[25] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pages 171–188, 2009.

[26] Stefan Tillich and Christoph Herbst. Attacking State-of-the-Art Software Countermeasures-A Case Study for AES. In *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, pages 228–243, 2008.

[27] Kris Tiri and Ingrid Verbauwhede. A VLSI Design Flow for Secure Side-Channel Attack Resistant ICs. In *Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*, pages 58–63, 2005.