Reactive Side-channel Countermeasures: Applicability and Quantitative Security Evaluation

Giovanni Agosta[a], Alessandro Barenghi[a], Gerardo Pelosi[a,*], Michele Scandale[a]

[a]*Politecnico di Milano. Department of Electronics, Information and Bioengineering – DEIB. Piazza Leonardo da Vinci, 32. 20133 Italy*

Abstract

The security of cryptographic implementations running on embedded systems is threatened by side-channel attacks. Such attacks retrieve a secret key from a computing device observing the information leaking on unintended channels such as the energy consumed during a computation. The vast majority of the countermeasures proposed against such attacks aims at preventing the attacker from exploiting fruitfully the information leaking on the side-channel either altering it or hiding it within a higher noise envelope. Whilst all these countermeasures provide a quantitative security margin against an attacker, they do not provide an indication of having been successfully overcome, thus forsaking the possibility of taking a reactive action upon an eventual security breach. In an effort to propose a reactive countermeasure, we describe our proposal suggesting the introduction of redundant computations employing fixed fake keys (a.k.a. *chaff*s) to pollute the leaked information with plausible albeit deceitful one. We provide an in depth analysis of the proposed approach, highlighting the constraints to its effective applicability, and the boundary conditions which allow its employment for the securization of a system. We detail the attacker model considered, and the reactive security margin provided by the proposed scheme, highlighting the extent of the realizability of a reactive countermeasure, given the nature of the side-channel information. To provide experimental backing to our analysis, effectiveness and efficiency results on the Advanced Encryption Standard (AES) cipher implementation as well as lightweight block ciphers implementations running on an ARM Cortex-M4 processor are shown.

Keywords: Applied cryptography, embedded systems security, computer security, automated countermeasure application, reactive countermeasures

1. Introduction

In recent years, the use of cryptographic primitives became essential for embedded systems due to their widespread adoption in several security-sensitive domains such as health-care, automotive and industrial control. Modern cryptographic systems are designed to withstand both protocol-level and mathematical cryptanalyses; as a consequence, attackers often focus on the analysis of the side effects of the computation performed by the device. There is a vast corpus of academic and industrial literature proving how the physical access to an embedded device may enable the recovery of sensitive information (typically, the secret key employed in a cryptographic algorithm), which is otherwise supposed to be hidden, through exploiting the side-channel leakages of the underlying computing platform [1–3].

Among the possible cryptanalyses employing the side-channel leakage, Differential Power Analysis (DPA) and Differential Electro-magnetic Analysis (DEMA) are very well known [3, 4]. Both analyses follow a common work-flow: first they record a measurement of either the power consumption or the electro-magnetic (EM) emissions of the target device (the measurements being known as *power-* or *EM-traces*) for a large number of runs with different input values. Subsequently, they select an intermediate operation of the algorithm employing a part of the secret key, and compute a consumption/emissions prediction for every possible value of the secret key portion, according to a model of the triggered switching activity (e.g., the Hamming weight of the output of the operation). Finally, the predicted consumption/emission values are statistically matched against each sample of the recorded power/EM traces to assess which key hypothesis yields the prediction fitting best the actual measurements. In this fashion, the secret key can be recovered, one part at time, even if the relevant information is stored within the device in a non accessible way.

DPAs and DEMAs have been proven practically viable against production grade implementations of ciphers, both in hardware and software, even with an inexpensive equipment. A significant amount of research effort has been directed to devise effective and efficient countermeasures. The most adopted strategies to design countermeasures focus on the principles of *masking* [3, 5, 6], *hiding* [3], and *morphing* [7–9] and aim at reducing the effective side-channel leakage. Masking aims at invalidating the link between the predicted hypothetical emission/power consumption values (bound to the selected intermediate operation) and the actual values processed by the crypto-

*Corresponding author

Email addresses: giovanni.agosta@polimi.it (Giovanni Agosta), alessandro.barenghi@polimi.it (Alessandro Barenghi), gerardo.pelosi@polimi.it (Gerardo Pelosi), michele.scandale@polimi.it (Michele Scandale)

graphic primitive. In a masked implementation, each sensitive intermediate value is concealed through splitting it in a number of shares, which are then separately processed. Hence, the cryptographic primitive is modified to correctly process each share and recombine them only at the end of the computation. Hiding methods aim at concealing the relation between the emission/power consumption of the device and the operations performed by the cryptographic primitive. In software cipher implementations, these strategies are based on execution flow randomization via instructions rescheduling (e.g., permuting the sequence of accesses to look-up tables) and/or on inserting random delays made of dummy operations. The morphing technique prevents an attacker from being able to construct a reliable model of the side-channel behavior of the device, changing how a cryptographic primitive is computed at each execution of the algorithm. The first proposed technique to achieve this combines the implementation of the chosen cryptographic primitive with a polymorphic engine which dynamically re-writes the binary code of the sensitive instructions to be protected, at runtime [7]. This strategy enables the generation of many different versions of the protected code at the designer's will, at each run of the cipher, thus preventing any attacker both from recognizing the exact point in time where the selected intermediate operation is executed, and from understanding how such operation is actually computed. The former effect can be classified as hiding-in-time, while the latter one prevents the formulation of a proper consumption model. Tackling the issue of platforms where the code memory is not writable, the approach described in [8] picks at random among a set of semantically equivalent code fragments to obtain different execution paths at each run of the same protected algorithm implementation.

All the aforementioned countermeasure strategies aim at either reducing the actual side-channel leakage or preventing its exploitation altogether, as the prime and only way to hinder side-channel attacks. None of the aforementioned strategies actually leave the leakage intact and blend it within a crowd of plausible, albeit fake, pieces of information. By contrast, it is commonplace, and a well established practice, in the network and system security communities, to provide intentionally vulnerable, worthless targets to the adversaries [10]. The purpose of such targets is to act as a red herring for adversaries, enabling the detection of malicious intentions towards the networks and hosts owned by the legitimate users (e.g., intrusion attempts). Other notable examples of using decoy resources to detect security breaches is found in the practice of deploying either phony credentials (such as credit card numbers) to discover their theft, or fabricated documents to act as bait for possible inside adversaries aiming at violating the system's usage policy [11].

The focus of this work is to provide an in depth description and security analysis of the reactive defense strategy presented in [9], 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 exploiting the information leakage during the decryption of a ciphertext. The attacker is assumed to have no means of access to the output of the decryption on the device, can only observe the actions performed by the attacked security system, and should not be able to distinguish an incorrectly decrypted plaintext from a correct one without interacting with the system itself. Practical application scenarios include keyless entry systems, physical authentication systems based on the transmission of an encrypted cryptographic token, or broadcast authentication schemes for wireless sensor networks. The *chaff countermeasure*, proposed in [9], swarms the attacker with dummy side-channel leakages among which the real one is blended. This allows the system designer to detect side-channel attacks whenever the adversary employs an incorrect value to produce forged encrypted content to be fed into the attacked system. This, in turn, allows a prompt response to the breach attempt before the attack succeeds, a much welcome feature in domains such as automotive, sensor networks and industrial control. We note that, since the fake leakage is not distinguishable from the real one, the security of the proposed defense strategy is not altered even when pitted against profiled cryptanalyses, as the fake leakage will provide the same information of the correct one to them. This is in contrast with leakage suppression techniques proposed in the current state-of-the-art (e.g., masking and hiding), where the defender attempts at hindering the exploitation of the leaked information raising the required technical effort to lead the attack.

We describe the application scenario of such a countermeasure, pointing out the applicative context and attacker model, and providing a hint of how the recently proposed *Honey Encryption* scheme [12, 13] may be employed to broaden the number of viable application scenarios where the chaff countermeasure can be applied; thus, expanding and improving the discussion with respect to the one in [9]. We report how to automatically apply the proposed countermeasure to software cipher implementations, and implement it as a transparent compiler pass in the LLVM compiler suite. Subsequently, we provide a new and detailed security analysis, highlighting the extent of the protection provided by the use of *chaff*-keys both against a single side-channel attack, and against a cascade of attacks. To practically ground our security analysis, we report experimental results on execution time and code size overheads due to the introduction of chaff-based countermeasures on a range of block ciphers, suitable to the use case scenario for which the chaff countermeasure is proposed. We provide results on the AES cipher, as it is a widely used standard employed also in current high security keyless entry systems (e.g., NXP ACTIC-4G [14], Atmel ATA5795C [15]), and was analyzed in [9], albeit with a lower number of chaff keys. We extend our experimental campaign, with respect to the one in [9], considering the computing requirements and code sizes of the lightweight ciphers XTEA [16], Speck 128/128 [17], Simon 128/128 [17], Present and Clefia. Present and Clefia are the ISO standard lightweight ciphers [18].

The rest of the article is organized as follows: in Section 2 we point out the scenario and the applicability conditions of the chaff-based countermeasure, as well as the attacker model. In Section 3 we highlight the property of a chaff-protected implementation and describe the methodology and tool to automatically apply the countermeasure. In Section 4 we detail the security analysis of the chaff-based approach, while in Section 5

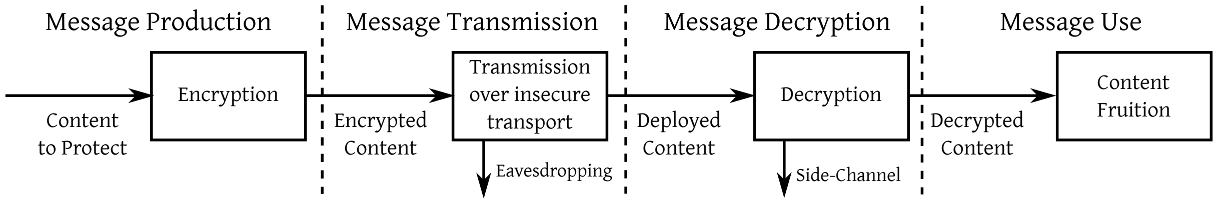


Figure 1: Information flow in the typical scenario where the side-channel chaff countermeasure can be applied. The attacker may gain knowledge of the data in the message transmission phase directly via eavesdropping, and try to retrieve the secret key in the message decryption stage via side-channel attacks. The message production and message use stages are considered to be safe

we provide our experimental evaluation, highlighting the advantages of applying the chaff-countermeasure to lightweight ciphers. Finally, in Section 6 we draw our conclusions.

2. Scenario

The scenario where the reactive chaff-based countermeasure is deployed is the one where the system manufacturer desires the deployed equipment to take an active stance against side-channel adversaries through reacting to an attack attempt, e.g., signal to the data owner the malicious action or wipe the secret key in the device.

The typical information flow of the scenario where it is possible to put into being a reactive countermeasure is the one represented in Figure 1. In this setting, a message produced in a safe environment and encrypted by means of a symmetric cipher, is sent to the device targeted by the side-channel attacker. The encrypted content is sent over a transport layer, which is assumed to be completely open to the attacker's eavesdropping actions. The message confidentiality during transmission is provided by the symmetric encryption layer, while unintentional transmission errors are prevented by an error correction code applied to the encrypted message. The content is decrypted by the deployed device into which the secret key is stored by the manufacturer in a non-extractable fashion. This decryption action is the actual target of side-channel attacks, with the final purpose of employing the derived key to forge fake contents to be fed into the device. In this context, the augmentation of the side-channel information with fake leakage induced by chaff keys may induce an attacker to employ an incorrect key to forge the contents to be fed to the device, thus enabling detection. We note that the augmented decryption activity on the attack-target side does not require any change in the rest of the content delivery chain; namely no extra data should be enciphered and sent, save for the correct plaintexts. V In the following, we explicitly state the capabilities of a side-channel attacker in the reported scenario and the applicability conditions for a proper instantiation of a chaff-based countermeasure.

Attacker Model. The attacker knows the complete details of the whole implementation, i.e., has full access to the binary code of the cipher running on the platform, save for the value of the secret- and the chaff keys. The attacker may intercept any communication relaying the enciphered content either while being transmitted over the network, or sent to the device by a neighboring storage, e.g., Flash memory. The attacker will try to retrieve the key piecewise exploiting side-channel leakage from the device, and may do so combining the results of one or more side-channel attacks. The attacker has no means of verifying whether a ciphertext decrypted under any given key (in particular, one retrieved via side-channel) is correct or not without direct interaction with the protected system.

The chaff-based side-channel countermeasure can be effectively employed as a reactive measure in the considered communication scenario, with the specified attacker model, if three general conditions are met by the system.

Applicability Conditions. Condition *i)*: the result of the decryption primitive is not output by the destination circuit in any informative enough format. Condition *ii)*: the use of the decrypted plaintext within the implementation does not leak information on the side-channel, which can be exploited to determine it. Condition *iii)*: the plaintext format is such that it is not possible for an attacker to distinguish a correct decryption outcome from an incorrect one relying on features of valid plaintexts. For example, it is not possible to distinguish the correct decryption outputs, relying on them not being uniformly distributed over all the possible bit-strings with the same length.

It is worth noting that the last condition is fundamental to instate a reactive countermeasure, as it forces the attacker to interact with the system without checking, in advance, the validity of a set of candidate keys on a different computational platform. A practical scenario, satisfying the aforementioned conditions, is the one where a transmitter encrypts a randomly and uniformly generated plaintext, which is decrypted by the receiving end, and employed as an authentication token to perform an action, e.g., unlock a door in a keyless entry system, or execute a broadcast authenticated command on sensor network nodes. Indeed, condition *i)* is satisfied as the receiving and decrypting device does not output the value of the plaintext at all. Condition *ii)* can be met protecting the comparator matching the decrypted plaintext with the expected one. This can be effectively and efficiently achieved with common, proactive, side-channel countermeasures such as Boolean masking, since the computation to be performed is remarkably small. Finally, condition *iii)* is satisfied as the correct randomly generated code cannot be distinguished from an incorrect decryption outcome by the attacker.

Among the applicability conditions of the chaff-based countermeasure, the third condition is often the most stringent one with respect to the possibility of applying the reactive countermeasure. However, it is possible to broaden the spectrum of the application of reactive countermeasures, thanks to the recent proposal of the so-called Honey Encryption (HE)

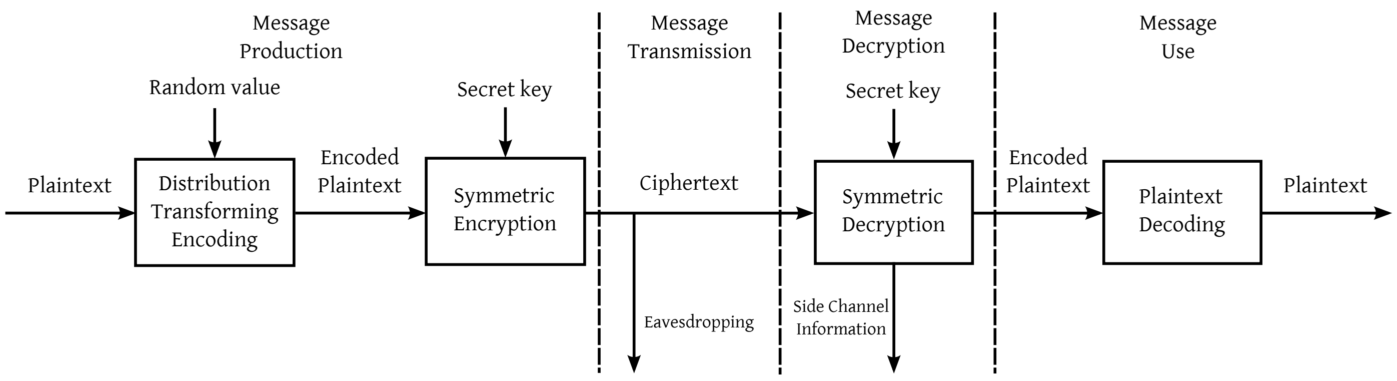


Figure 2: Information flow of a reactive countermeasure amenable scenario involving the use of the honey encryption scheme. The attacker is assumed not to be able to extract any knowledge from the message use phase, in the same fashion as the common scenario

scheme [12, 13]. The scenario where the chaff-based countermeasure can be deployed is extended as depicted in Figure 2. The honey encryption scheme performs a randomized encoding of the plaintext with a Distribution Transforming Encoding function (DTE), resulting in an encoded plaintext, which is uniformly distributed among all the possible bit-strings with the same length. Such an encoding takes as an ancillary input a random value so to allow the encoding of the same plaintext to take different values at each time. The encoded plaintext is subsequently encrypted by means of a symmetric cipher, employing a secret key and, finally, transmitted. The decryption primitive of the HE scheme starts by deciphering the received ciphertext, and proceeds to perform a deterministic decoding procedure on the result. The decoding procedure is designed so that it outputs a value which can be plausibly assumed to be a valid plaintext for any value fed to it. In [13], the authors provide practical constructions for the DTE and the decoding stage, specialized for the encryption of credit card numbers, PINs, CVVs and RSA secret keys. Thus, considering the proposed constructions, an HE scheme enables a generic scenario (as the one in Figure 2) to satisfy the condition *iii)* from the applicability conditions of a reactive countermeasure. To prevent the attacker from exploiting the side-channel leakage which the implementation will produce during the HE decoding and “message use” phase, it is possible to protect them with conventional proactive side-channel countermeasures, thus fulfilling also the applicability conditions *ii)*. We also note that a, possibly computationally cheaper, alternative to the protection of the message HE decoding stage with proactive side-channel countermeasures is to perform it for all the decryption outputs, both the correct one and the ones obtained with chaff keys. If such a solution is chosen, the side-channel information leaked by the HE decoding stage will show that all the plaintexts, both the correct one and the chaffs are being treated equally.

3. Chaff countermeasure

Under the assumption that both the scenario and the attacker model considered by the designer fit the ones described in the previous section, it is possible to provide a reactive side-channel countermeasure by means of implementing the block cipher to be protected so that its side-channel leakage satisfies the following property.

Chaff Leakage Property. The countermeasure adds to the chosen side-channel leakage one or more chaff leakages, i.e., it intentionally leaks information on the side-channel behaving as if one or more incorrect keys were actually being employed to perform the encryption. The effect of the observed information leakage is to have the side-channel attack matching more than one key as the correct one, without providing any hint to which ones are fake. In particular, the correct key should not be distinguishable either comparing the goodness-of-fit of the correct hypothesis to the actual measured side-channel against the one of the fake keys, nor through performing inferences on which instant in time is characterized by said goodness-of-fit.

To achieve this, the computations employing the fake keys should mimic exactly the correct one for all the portion of the cipher subject to side-channel attacks. This in turn implies that, for each attempt to retrieve a portion of the secret key, the attacker will in fact obtain both the correct value and one or more incorrect ones, which will appear equally likely considering the results provided by the performed side-channel analysis. However, executing sequentially replicas of the cipher instructions is not sufficient to fully blend the behavior of the chaff keys into the one of the real one. It is in fact typical during the attacks to compute the fitness of the key-dependent hypothesis to the measurements in a time-wise fashion, effectively obtaining goodness-of-fit values for each time instant. Thus, the attacker is able to distinguish when the *key hypothesis* fits best the behavior of the device and, employing the information coming from the knowledge of the cipher algorithm (code), bind the time instants in which the hypotheses fit to the executed instructions of the cipher. Finally, checking on the cipher code which instructions are employing the correct key, the attacker will discard the hypotheses related to the chaff keys safely.

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, we resort to an execution trace randomization technique similar to the one described in [8] so that either the instruction computing the real cipher result, or one of its chaff is executed randomly at each cipher run. Such a ran-

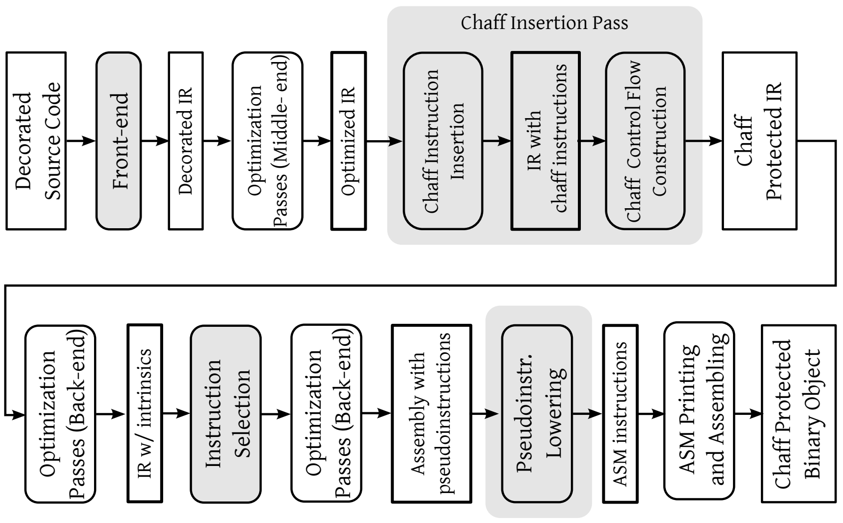

Chaff Insertion Pass
Decorated Source Code
Front-end
Decorated IR
Optimization Passes (Middle- end)
Optimized IR
Chaff Instruction Insertion
IR with chaff instructions
Chaff Control Flow Construction
Chaff Protected IR
Optimization Passes (Back-end)
IR w/ intrinsics
Instruction Selection
Optimization Passes (Back-end)
Assembly with pseudoinstructions
Pseudoinstr. Lowering
ASM instructions
ASM Printing and Assembling
Chaff Protected Binary Object

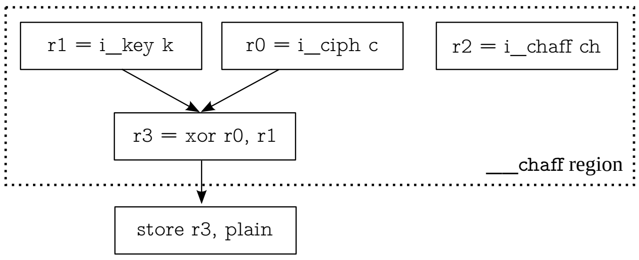
Figure 3: LLVM compiler pipeline with the modifications highlighted in grey, and the new passes highlighted by larger gray boxes. Rounded boxes depict (sets of) compiler passes, while square ones represent the data being processed

domized scheduling causes the fitness for multiple keys to peak simultaneously, since the side-channel analysis needs to combine a statistically significant amount of measurements from different runs together to compute the fitness of the hypotheses time-wise. The execution trace randomization will thus select, through a Random Number Generator (RNG) 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. The random choice of the execution path guarantees that all paths are eventually taken. 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 (#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 #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.

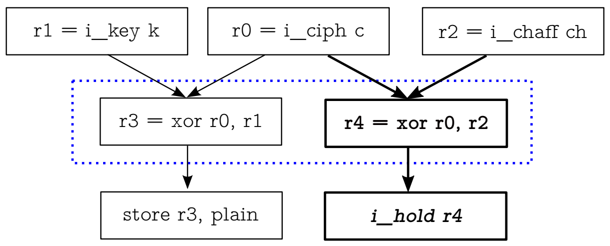
3.1. LLVM Compiler Pipeline Modifications

We now describe how to perform the automated application of the chaff countermeasure to a software cipher implementation by means of two new compiler passes and some modifications to the ones in the LLVM compiler pipeline, as highlighted in Figure 3. The modifications concern the *front-end* of the compiler and the *instruction selection* pass in the compiler backend, while the two new passes perform the *chaff countermeasure insertion*, and the *pseudo-instructions lowering*. In this description we will say, according to the common compiler technology terminology, that an instruction "defines" the variable to which its result is assigned, and "uses" the ones which appear as its inputs.

Front End Modifications. The front-end modification allows the developer to specify which portion of the cipher should be protected with the proposed countermeasure, and tag the variables containing the true key, plaintext, ciphertext and chaff keys. To this end, the compiler frontend has been extended to support five custom attributes. The keyword `__chaff` is prefixed to a `C` code-block (i.e., a sequence of statements enclosed between curly braces) to mark it as the portion of cipher where the chaff countermeasure should be applied. We note that it can be advantageous in terms of performance, although with no security drawbacks, to protect only a portion of the block cipher, as stated in [8], hence the freedom for the programmer to choose to do so. The keywords `__ptx`, `__ctx`, `__key` are employed to mark the plaintext, the ciphertext and the key values, respectively. The attribute `__chaff_key( <key_length>, <num_chaff>)` marks the variable containing the chaff keys and provides their number. The attributes tagging variables are lowered into custom *intrinsic instructions* without side-effects, which use the tagged variable and define a copy of it. In this way, the tagging information is transparently preserved up to the point of the middle-end where the chaff insertion pass acts. The code block identified syntactically via the `__chaff` keyword, is lowered into an LLVM *code region* employing a method analogous to the one in [8]. The net effect is that no code can be hoisted out of, or sank into the marked code region by the middle-end optimizer passes. The LLVM Intermediate Representation (IR), extended with intrinsics and code regions, is fed to the compiler middle-end. The decorated IR is processed by all the middle-end optimization passes yielding the optimized IR employed as input by our chaff insertion pass. For the sake of clarity, we will describe the inner working of the chaff insertion pass as two separate stages, as depicted in Figure 3: *Chaff Instruction Insertion* and *Chaff Control Flow*



(a) Optimized IR



(b) IR with chaff instructions

```
          i__key r1, k
          i__ciph r0, c
          i__chaff r2, ch
          load r5, rnd
          and r5, r5, #1
L__switch: jmp [ L__table, r5 ]
 L__table: L__case1
          L__case2
 L__case1: i__xor r4, r0, r2
          i__xor r3, r0, r1
          jmp L__end
 L__case2: i__xor r3, r0, r1
          i__xor r4, r0, r2
          jmp L__end
   L__end: store r3, plain
          i__hold r4
```

(c) Assembly with pseudoinstructions

Figure 4: Depiction of the intermediate representation of the code of a toy cipher composed of a single `xor` operation, during its processing by the modified LLVM compiler. In (a) is reported the IR after the middle end optimizer has completed its work, in (b), the one after the chaff instruction insertion has been performed, in (c) is depicted the low level IR right before the custom pseudo instructions are removed. Chaff instructions are represented in boldface, intrinsic instructions are prefixed by `i_`, blue instructions constitute the additional control flow. Instructions in italics provide artificial uses so that the *dead code elimination* optimization passes do not remove the chaffs

Construction. We also recall that the LLVM IR is expressed in Single Static Assignment form, i.e., each instruction defines a new variable, represented as a virtual register, save for `store` and `jump` instructions. As a consequence, it is commonplace to identify an instruction with the variable it defines, whenever possible. We will use as a running example a trivial cipher where a single input variable is decrypted via a single `xor`. Figure 4(a) provides a depiction of the state of the LLVM IR upon being fed to the chaff insertion pass, represented as its dataflow graph. The intrinsic instructions employed to mark the chaff, key and input ciphertext are represented as `i_chaff`, `i_key`, and `i_ciph`. The protected LLVM code region is enclosed in the dotted line.

Chaff Instruction Insertion. The chaff instruction insertion needs to compute, as its first step, the set of instructions $\mathcal{K}$ containing all the ones which use, directly or indirectly at least a variable derived from the cipher key. This can be done following the definition-use relation chains over the program IR, and adding all the nodes belonging to the chains starting with the intrinsics inserted by the front-end. We note that the set $\mathcal{K}$ contains both the instructions computing the cipher and the ones manipulating the keys before its combination with the input, i.e., the code portion also known as the *keyschedule*. For the sake of clarity, the reported running example does not have a key schedule; as a consequence $\mathcal{K}$={`i_key`, `xor`, `store`}. Similarly, the set of all the instructions which use the input ciphertext C, either directly or indirectly, is computed. This set encompasses all the instructions in the cipher computation, including possibly some which do not interact with the cipher key (e.g., the initial permutation in the DES cipher). In the example, C={`i_ciph`, `xor`, `store`}. Once $\mathcal{K}$ and C have been built, it is possible to locate the cipher instructions that will be the target of side-channel attacks as the ones in $\mathcal{K} \cap C$, i.e., the ones combining variables derived from the input with the key. To insert chaff keys during the whole computation, the instructions belonging to $\mathcal{K} \cap C$ and using only variables not in $\mathcal{K} \cap C$, indicated from now on as the *roots* of $\mathcal{K} \cap C$, are identified. The roots are the starting point from which the countermeasure should be applied on the cipher. In the example, there is a single root, the `xor` operation, which is also the only node in $\mathcal{K} \cap C$ contained in the `__chaff` region. Subsequently, the chaff instruction insertion pass replicates all the instructions belonging to the keyschedule, using the chaff keys as their inputs, and, for each duplicate, it memorizes which original keyschedule instruction is bound to the corresponding chaff in a map $\mathfrak{M}$. This map is employed, throughout the whole pass, to track the binding between original instructions and chaff ones. In our example, the keyschedule duplication phase does not have any effect, as there is no keyschedule. Once a keyschedule, identical to the original one save for the processed data, has been inserted for each chaff key, the pass proceeds to generate chaff instructions for the whole portion of $\mathcal{K} \cap C$ enclosed in the LLVM code region specified by the programmer. This step is performed acting on a working queue of instructions, which is initialized with the roots of $\mathcal{K} \cap C$, until the fix-point where the queue is empty is reached. For each one of the elements, the pass duplicates it, taking care to replace the uses of the original instruction coming from correct-key derived variables, with the corresponding ones from chaff-keys, as stored in the map $\mathfrak{M}$. Figure 4(b) reports the added chaff-`xor`, in boldface type, with its correct-key use `r1` replaced with the corresponding chaff-key value `r2`, alongside its original counterpart ⟨`r3=xor r0,r1`⟩$\in \mathcal{K} \cap C$. After this step, the map $\mathfrak{M}$ is updated with the binding between

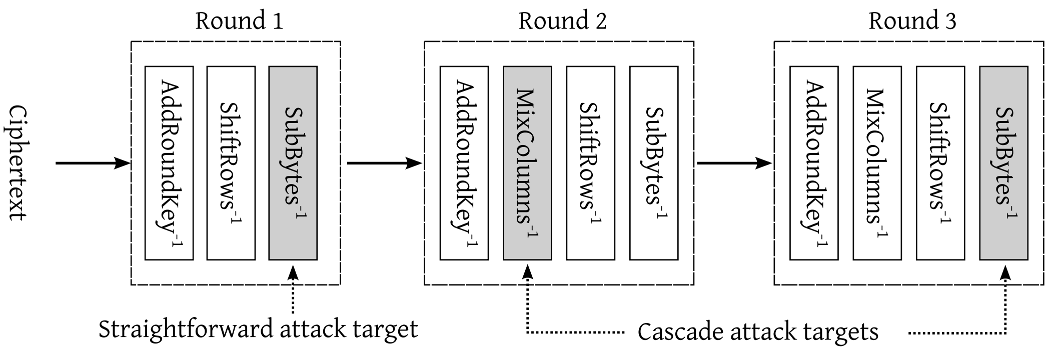


Figure 5: First three rounds of an AES-128 decryption computation. The operations computing the intermediate values targeted by the straightforward attack and the cascade one are highlighted in gray

the freshly inserted chaff instruction and the original one, so to represent the relation between the values defined by them. In our example the new relation is represented in Figure 4(b) by the blue dotted line binding the original and the chaff-`xor`. As the last action, all the instructions using the value defined by the original instruction, within the region to be protected, are added to the working queue. In the example, no such instructions are present, as the only use of ⟨`r3=xor r0,r1`⟩ lies outside the `__chaff` region. In case there are no such instructions, the pass takes an extra action to prevent the *dead code elimination* passes from removing the added chaff instructions. In fact, in the current state, the result of such chaff instructions (⟨`r4=xor r0,r2`⟩ in the example) is never used, thus marking them for removal as dead code. To this end, the pass inserts an intrinsic, defined to have generic memory side-effects on the computation (named `i_hold` in Figure 4(b)), which uses the results of all the chaff instructions which were just inserted. From now on, we represent both intrinsics and pseudo-instructions in our examples prefixed with a `i_`, with a small notation abuse. Since such an intrinsic cannot be optimized due to its unpredictable side effects, and uses the variables defined by the chaff instructions, they will no longer be considered dead code.

Chaff Control Flow Construction. The IR with the inserted chaff instruction is subsequently processed by the chaff control flow insertion pass (Figure 3) where the actual RNG-driven selection construct is created. To this end, the pass builds a switch-case like control structure for each each relation stored in the map $\mathfrak{M}$. Figure 4(c) reports the assembly with pseudo-instructions which results from this construct after the instruction selection pass, for the sake of clarity. All the labels and instructions related to the chaff control flow are highlighted in blue. Two constraints must be tackled in building the switch-case structure: all the execution paths must take the same time, and no rescheduling of the instructions must be made. To achieve a constant execution time on all the paths, the structure of the switch case construct is built with a *jump-table* strategy. The jump table strategy enumerates the target addresses of the first instruction of each case (`L_case1` and `L_case2` in our example) in a constant table placed within the code (tagged by the label `L_table` in Figure 4(c)). The random choice among the cases is driven by a random integer (stored in `r5` in our example) which is reduced modulo the number of cases, and subsequently employed as the offset in the branch table to compute the target address by an unconditional indirect jump. The body of each case must be built out of instructions which must not be rescheduled, hoisted, sank or eliminated by the *common subexpression elimination* pass. To this end, the chaff control flow construction substitutes each of them with a corresponding intrinsic, using and defining the same values, and declared to have unmodeled side-effects on the computation. This allows the pass to build the case code blocks enforcing the desired instruction order, so to achieve the indistinguishability in time effect. In the example, the `i_xor` pseudo-instructions corresponding to the lowering of the `i_xor` intrinsic are reported.

Instruction Selection and Pseudo-instruction Lowering. The IR decorated with the custom intrinsics by the chaff insertion pass is subsequently processed by the backend optimization passes up to the *instruction selection* pass. Since the IR contains our custom intrinsics, which are designed to be opaque, and thus cannot be recognized by the instruction selection pass as representing any of the instructions of the underlying ISA, we modified the pass so that each of them is lowered into a corresponding pseudo-instruction. After the customized instruction selection pass, the remaining backend passes (e.g., register allocation) are run, bringing the representation closer to valid assembly for the target architecture. The resulting machine instruction list, reported in Figure 4(c), is then processed by a custom pseudo-instruction lowering pass, which maps all the ones corresponding to actual machine instructions back into their correct format, while removing the placeholder ones aimed at avoiding dead code elimination. In the example, the `i_xor` is translated into an actual `eor` for the ARM ISA, while the marker pseudo-instructions `i_key`, `i_ciph`, `i_chaff` are removed or substituted with `mov` instructions depending on the register allocation. The `i_hold` place-holder is simply removed. In this stage, it is also checked that all the exit jumps from each case of the switch-case constructs (the ones to `L_end` in the example) were emitted, as it is a commonplace optimization to omit the last one, due to the natural execution flow falling through to the next instruction anyways. Since this optimization is detrimental to having all the cases with a constant time execution, we insert the missing jumps whenever needed.

4. Security analysis

We provide the quantitative security analysis of the chaff countermeasure against a simple attack (as the one considered in [9]) and against a cascade attack strategy able to reduce the uncertainty on which side-channel extracted key is the correct one down to the minimum, i.e., picking one out of #chaff+1 possible keys. To illustrate the steps of both attacks, we will employ as our running example an instance of AES cipher with a 128-bit key, of which the first three rounds are depicted in Figure 5. We start by recalling that the general framework of a passive side-channel attack (SCA) retrieves the whole secret key K piecewise guessing the side-channel behavior of the device during a precise time instant where an operation involving a p-bits large subset of the bits of K is performed. The side-channel behavior of such a time instant is predicted for all the 2^p possible values of the small portion of K under consideration. The correct prediction is distinguished from the other ones by means of a statistical test gauging the goodness of fit of each of the predictions against the measurements time-wise [3]. The entire value of K is put together after retrieving it piecewise in $\frac{K}{p}$ different analyses performed on the same sampled data from the side-channel, while changing the targeted intermediate operation to guess different key bits. In our running example involving the AES-128 cipher, an attacker may retrieve the whole AES key in 8-bit portions, thus leading $\frac{128}{8}$=16 attacks on the same measurements set. The recalled framework is known in literature as a *first order* SCA as it exploits the knowledge of a single time instant where an operation takes place to validate the side-channel behavior prediction. More sophisticated attacks exist where, out of the need of overcoming proactive SCA countermeasures, the attacker employs the values measured on the side-channel in more than one operation of the cipher, leading to the so-called *high-order* SCAs. In the following, we will focus on first-order SCAs under the assumption that the implementation is protected only with the reactive chaff-based countermeasure. An analogous line of reasoning can be followed assuming that other, proactive countermeasures are in place, and the attacker is bypassing them via an high-order attack.

Straightforward Attack. Let c be the number of chaff keys employed by the considered cipher implementation, protected with the methodology described in Section 3. Upon following the aforementioned first order SCA procedure, a set of c+1 plausible values for each one of the $n=\frac{K}{p}$ key portions is retrieved by the attacker. To this end, the attacker will need to employ a computational effort which is $\Omega(n{\cdot}2^p{\cdot}t)$, where t is the amount of samples gathered by the measuring equipment [3]. We note that, this is a strict lower bound where the attacker is able to compute the statistical distinguisher of choice in constant time, which is not usually the case. Once the n sets of key portions are retrieved, the attacker needs to successfully reconstruct the correct cipher key K picking the correct portions. The value of p is chosen by the attacker, taking into account the fact that increasing it implies an exponential increase in the amount of computational effort required to lead the SCA, while decreasing significantly the number of possible keys to be tried. In our running example, n is commonly picked to be 16 as the adversary tries to extract the key byte-wise (i.e., p=8). An example of targeted result of an intermediate computation for such an SCA is the table look-up action performed by the SubBytes^{-1} primitive of the first round of AES as marked in Figure 5. Such a computation depends exactly on 8 key bits, and thus requires 2^8 consumption hypotheses to be made for each input. The lowest publicly reported value for n is n=4, in case of an attack against an AES-128 cipher retrieving p=32 key bits at once [19].

Once the possible candidate values for each key portion are collected, the attacker will need to piece them together, with no information to distinguish the chaff ones from the correct ones within each set of collected key portions. However, we note that the attacker is in knowledge of the order in which the candidate values are to be taken from the sets, as they are bound to the intermediate value predicted during the attack, identifying which portion of both the correct and the chaff keys was retrieved. As a consequence, the number of plausible values for the entire cipher key is $(c+1)^n$. Since the attacker does not have any kind of decryption oracle, exploiting the information retrieved up to now (in this straightforward attack) would require checking which one of the $(c+1)^n$ keys is correct interacting with the system [9]. Considering values of $p{\in}\{8,16,32\}$, $n{\in}\{16,8,4\}$ and c=3 for our running example, we can conclude that the attacker will succeed, in guessing the cipher key with a single attempt, with probability 2^{-32}, 2^{-16} and 2^{-8}, respectively.

Cascade Attack. Despite the straightforward attack strategy being effective [9], it is possible for an attacker to further increase his advantage in guessing the correct key among the chaff ones. Indeed, employing a cascade of first order attacks, each one relying on the results of the previous one, the number of candidate values for the entire cipher key can be reduced from $(c+1)^n$ down-to c+1.

The key idea of the cascade attack is to exploit the natural data dependencies present in the cipher to validate the correct combination of a set of key portions (which is potentially smaller than the entire key), thus discarding combinations mixing correct key portions with chaff key portions.

To this end, assume an attacker has already executed a straightforward attack obtaining n sets of candidate key portions $S_1, S_2, \ldots, S_n$, each one c+1 elements wide. Exploiting the fact that any block cipher computation involves an increasing portion of its cipher key (as its computation proceeds down the rounds of the algorithm [5]), the attacker targets an intermediate value further down the cipher computation, chosen as it is depending on the values of $1{<}p'{\leq}n$ candidate key portions picked from $S_1, S_2, \ldots, S_{p'}$.

To the end of predicting the side-channel behavior of the said (second) intermediate value computation, the lowest number of hypotheses for the involved key portion is $(c+1)^{p'}$, as one (sub-)portion for each of the p' involved should be picked from the proper sets S_i. The outcome of the attack will be c+1 plausible values for the key portion involved, which in turn will be $(p'{\cdot}p)$-bit wide. Iterating such an attack will yield $n'{=}\lceil\frac{n}{p'}\rceil$ new sets of candidate key portions, lowering the amount of valid values for the entire keys to $(c+1)^{n'}{<}(c+1)^n$.

The reason for the number of plausible values being reduced is the fact that only correctly combined key portions will yield a value upon which sensible intermediate values may be predicted. This, in turn, implies that only side-channel behavior hypotheses deriving from correctly combined key portions will show a correlation with the actual measured one, as no computation employing a mixture of correct and chaff key bytes is performed, and thus no such information leakage is present.

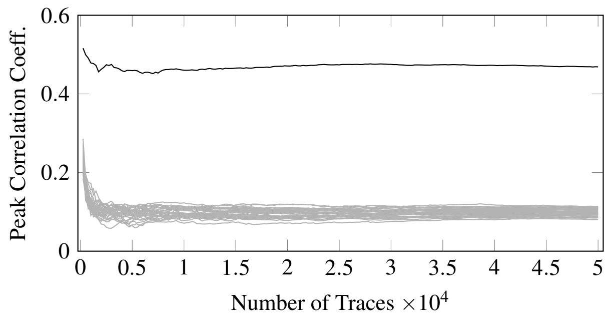
Continuing our running example, assume the attacker has already performed a straightforward attack with p=8, retrieving n=16 sets of (byte-wide) candidates for the key portions of the considered AES-128 cipher implementation. The attacker chooses to target one output byte of the MixColumns operation present in the second decryption round of the AES. One such output byte depends on the values of four key portions among the ones retrieved in the first attack, thus p'=4 in this case. Considering three chaff keys are employed, c=3, the attacker will formulate $(3 + 1)^4$=256 different hypotheses, each one 4-byte wide, among which only the c+1=4 made out of bytes belonging from the same key will be predicting the side-channel behavior of a part of the computation correctly. As a consequence, $n'=\lceil \frac{16}{4} \rceil$=4 candidate sets are produced, leaving $(c + 1)^{n'}=4^4$=256 plausible values for the correct cipher-key.

The cascade strategy can be iterated to further reduce the number of plausible values for the cipher-key up to a single set containing c+1 candidate values. Such a reduction will be achieved as the attacker (once the computational effort is feasible) will be able to predict the side-channel behavior of an intermediate computation depending on the entire cipher-key. Therefore, he will observe matches against the physical measurements only for the correct values of both the cipher key and the chaff ones. In our running example, this can be obtained employing as the target of the side-channel attack, the output of the SubBytes primitive of the third decryption round (see Figure 5). Since its output depends on the entire cipher-key, comparing the predicted side-channel behavior for each one of the 256 plausible key values will report actual correlation with the measurements only c+1=4 times.

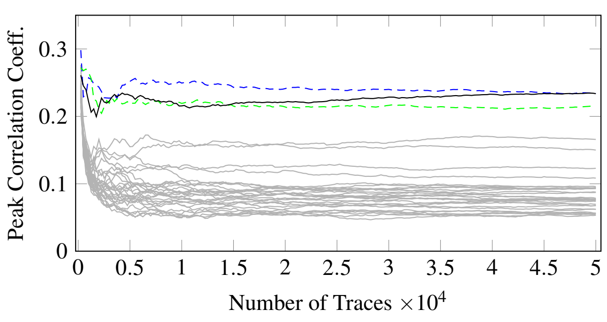
It is thus possible for an attacker, through a sequence of cascaded first order SCAs, to raise the probability of a successful key guess out of the information obtainable via side-channel up to $\frac{1}{c+1}$. We note that no further information on which one of the remaining key candidates is obtainable in the described attacker model, as there is no information concerning the computation of the value of the correct decryption alone available to the attacker. This is a tight requirement, as if a computation involving the said value alone were to leak information on the side-channel, a side-channel attack considering as target its output could be successfully set up (see Condition *ii)* in Section 2). Indeed, such an attack would reveal the value of the correct key alone, as only the correct plaintext is processed and employed in the device, while the ones coming from chaff key decryptions are typically discarded. We recall, as described in Section 2, that such an attack can be easily thwarted either protecting the computation employing the plaintext with conventional proactive countermeasures, or duplicating it also for the incorrect plaintexts. Finally, we note that the cascade first order SCA strategy works even in cases where the key material of a cipher is entirely composed of random values (i.e., the cipher has no keyschedule). Indeed, the capability of an attacker of progressively reducing the amount of viable candidates for the entire key value relies on the progressive combination of the correct key material with the state of the cipher during its computation. In particular, the described cascade attack strategy reduces the number of viable candidates for each guessed key portion to c+1 elements after each first order attack is performed. As a consequence, the cascade attack strategy is more profitable in ciphers where the operations between two "key additions" (i.e., the rest of the cipher round structure) yield a result where each one of the bits processed by them depends on all their inputs, a property commonly known as *diffusion*. A cipher achieving full diffusion within a single round will indeed present the attacker with the possibility of reducing the amount of candidates for the round key down to c+1 employing a cascade attack before any fresh key material is added to the computation. Since diffusion is a crucial property in secure block cipher design, the round of a block cipher is typically designed to maximize it, thus allowing the attacker to effectively come significantly close to the c+1 bound in practical cases. For instance, in our running example, the AES round allows the attacker to reduce the number of round key candidates to 4(c+1), as AES diffuses over a quarter of the round output values. The aforementioned strive of the cipher designers towards achieving a good diffusion in turn implies that the number of hypotheses to be made to extract each round key value will only grow by a small multiplicative factor over c+1 as a consequence of taking into account the values of the key material from previous rounds. We note that such factors do not accumulate over the number of rounds, as only actually performed computations will leak on the side-channel, thus allowing the attacker to deduce the correct combination of round key values following the same line of reasoning of the cascaded attack. Finally, we note that it is the indistinguishability in time among the operations acting on a correct key portion and its corresponding chaffs that constrains the attacker to resorting to a cascade attack strategy. Indeed, the apparently viable approach of computing c+1 times the same decryption primitive sequentially, once with the correct key and the remaining c times with chaffs (picking a key for each decryption primitive in random order), allows the attacker to exploit the information coming from the time-wise partitioning of the correlating measurement samples of the side-channel behavior to determine to which key a given portion belongs.

5. Experimental Evaluation

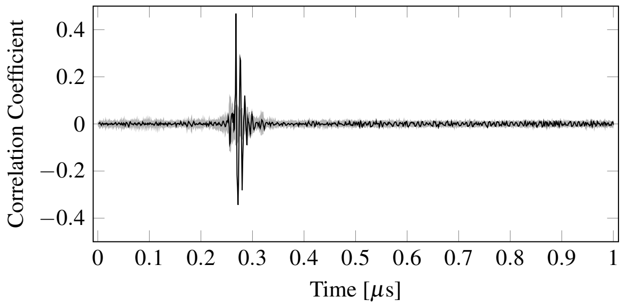
In this section, we provide the result of the experimental campaign both validating the effectiveness of the chaff countermeasure approach, and measuring its efficiency in terms of running time and code size overheads as a function the number of inserted chaff keys. The chosen platform for our evaluation is an STM32F407 microcontroller (μC) based on the ARM Cortex-M4 core with 192 kiB of SRAM and 1 MiB Flash memory mounted on a commercial grade 32F407CDISCOV- ERY board from STMicroelectronics. The core was clocked at 84 MHz during the whole experimental campaign. We chose to target a



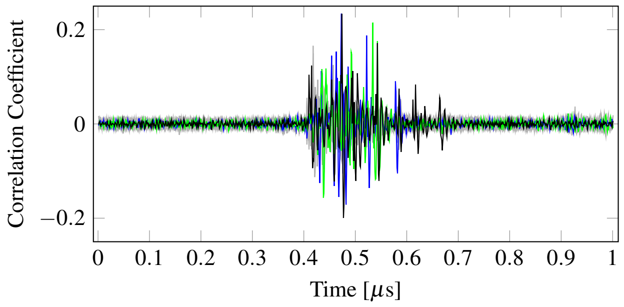
(a) Unprotected - Peak correlation coefficient vs. number of traces



(b) Protected - Peak correlation coefficient vs. number of traces



(c) Unprotected - Timewise correlation coeff. with 50k traces



(d) Protected - Timewise correlation coeff. with 50k traces

Figure 6: Results of a DEMA attack to the SubBytes primitive the AES (1 byte key hypothesis) both without and with two chaff keys, respectively. Correct key in black, chaff keys in green and blue, wrong keys in grey. Figures (a) and (b) report the peak correlation coefficient against the number of traces employed to compute it. The peak correlation coefficients of the chaff keys are close to the one of the actual key (with the differences determined by statistical artifacts) providing indistinguishability. Figures (c) and (d) report the timewise correlation coefficients computed employing 50k measurements of the EM emissions of the μC. The timewise correlation of the three keys (correct and two chaffs) take place in the same time interval, providing indistinguishability in time

32-bit μC as our experimental platform to follow the increasing trend towards the adoption of 32-bit μCs even in low-cost and power constrained environments [20–22]. All the `C` cipher implementations employed were compiled with a modified LLVM 3.4 compiler suite and release-grade optimizations (-O3) targeting the Thumb-2 instruction set.

The validation of the effectiveness of the approach has been conducted with both the instruction cache and data cache of the microcontroller, for the sake of obtaining a setup most favorable to the attacker. For the same reason, a standard compliant software AES implementation `C` based on a single S-Box was employed to minimize algorithmic noise. However, results similar to the ones reported were reported on the T-Tables based implementation of AES employed in both OpenSSL and PolarSSL, an SSL library specially tailored for embedded systems. The side-channel of choice for our validation are the electromagnetic (EM) emissions of the microcontroller during the computation of the cipher. The EM measurements were made by means of a loop probe obtained out of a 50 Ω coax cable, amplified through two stages of Agilent INA-10386 amplifiers, and sampled at 500 Msamples/s with a Picoscope 5203 DSO. The measurement trigger was driven by a signal coming from one of the μC GPIOs, which was raised at the beginning of each block cipher computation. The intermediate value chosen for carrying out the straightforward DEMA attack is a byte of the output of the first SubBytes primitive, as described in the running example of Section 4, and we employed its Hamming weight as the emission model. The implementation under attack was protected instructing the compiler to add two chaff-keys.

Figure 6 reports the results of the analysis of the EM emissions of the μC, both as a function of time and of the number of measurements performed. In particular, Figures 6(a) and 6(b) show how the correlation coefficient reported for both the correct key and the chaff ones is substantially the same, with the differences among them being determined by measurement noise in this instance of the side-channel measurement. In particular, the confidence intervals for two estimates are overlapping for any confidence level between 80% and 99.999%. Figures 6(c) and 6(d) report the evolution of the values of the correlation coefficients in time for both the unprotected and chaff-protected implementation respectively, showing a close-up view in time for the sake of clarity. Thanks to the randomization among different schedules of the correct and chaff computation, it is not possible for the attacker to distinguish the correct key from the chaff ones through exploiting a time-wise partition of the correlation peaks pointing to when the actual computation takes place. Note that the small difference in time between the unprotected and protected peaks of fitness in the plots is due to cropping issues, for clarity in representation.

The actual time difference is greater than the depicted one (which is around 2 clock cycles) due to the higher number of instruction being executed in the chaff-protected version. The results in Figure 6 thus show how the chaff property described in Section 2 is provided by the described implementation method.

Table 1: Execution times for the examined set of block ciphers as a function of the total number of keys c+1 employed. Dashed-out cells indicate that the resulting binary cannot be fit within the device Flash memory. Gray cells highlight the best performer for a given number of chaff keys employing the chaff-property providing technique (upper table). Timing results employing a naïve repeated cipher execution with different keys are reported for comparison (lower table)

	Time [μs]						
	baseline	chaff-based approach					
c+1	1	2	4	8	16	32	64
AES–T	17.17	111.5	168.5	308.9	678.4	1,572	–
Clefia	81.9	573.8	882.4	1,676	3,624	–	–
Present	168.4	499.2	896.6	1,928	3,993	8,356	–
Simon 128/128	23.7	61.35	258.5	505.0	996.6	2,009	4,174
Speck 128/128	15.11	28.82	70.46	235	433.8	874.1	1,894
XTEA	13.01	25.29	136.9	248.3	520.9	1,069	2,150

	Time [μs]						
	baseline	naïve approach					
c+1	1	2	4	8	16	32	64
AES–T	17.17	34.5	68.8	137.5	274.8	549.6	1,099.0
Clefia	81.9	164.6	328.4	656.0	1,311.2	2,621.6	5,242.4
Present	168.4	338.4	675.2	1,348.8	2,696.0	5,390.4	10,779.2
Simon 128/128	23.7	47.6	95.0	189.8	379.4	758.6	1,517.0
Speck 128/128	15.11	30.3	60.5	121.0	241.9	483.6	967.1
XTEA	13.01	26.1	52.1	104.2	208.2	416.4	832.7

After providing the experimental evidence of the effectiveness of the approach, we report the computational and code size overheads of chaff-based side-channel countermeasures, on a set of block ciphers including the Advanced Encryption Standard (AES); the ISO standard lightweight block ciphers [18]: Clefia and Present; the widely deployed reduced fingerprint cipher [16] XTEA; and the constrained environment block cipher proposals by US National Security Agency (NSA) [17]: Speck 128/128 and Simon 128/128. The choice of the ciphers was made targeting possible block cipher candidates for a small embedded system, fitting the scenarios previously mentioned in Section 2, where a reactive side-channel countermeasure provides an effective improvement over purely proactive ones. Timing measurements were made sampling a signal coming from one of the μC GPIOs, which was asserted at the beginning of the cipher computation, and de-asserted at the end of it. The GPIO was sampled at 500 Msamples/s with a Picoscope 5203 DSO, achieving a 2 ns resolution. All the timings are the result of an average of 30 executions of the primitive.

Table 1 reports the running times for all the benchmarked block ciphers, changing the number of chaff keys c to obtain a successful guess probability by the attacker in the 2^{-1}–2^{6} range when a cascade attack strategy is employed. Together with the timing obtained from the implementation employing the strategy described in Section 3, we also report the ones coming from iterating the entire cipher primitive, employing sequentially one of the c+1 keys for the sake of comparison. We recall that, while the former strategy forces the attacker to perform a "cascade attack", the latter does not provide indistinguishability in time of the power/EM traces related to the real and the chaff keys and will yield the same security margin of the chaff-based approach even if only a "straightforward attack" is carried out. Dashed out cells indicate that the resulting binary could not be deployed on the target platform as its size does not allow it to fit in the Flash memory of the target device. The results show how the standard AES block cipher implemented with T-tables provides sub-millisecond computation for all the considered choices up to 15 chaff keys (i.e., c+1=16), providing an effective solution for interactive load based computation on embedded systems. However, AES–T, Clefia and Present are affected by larger absolute overheads in computation time with respect to Simon 128/128, Speck 128/128 and XTEA. This behavior is to be ascribed to the round structure of the latter three cipher proposals, which is built out of a very reduced amount of modulo-2^{32} arithmetic and Boolean operations around a few word-sized variables. This results in a lower register pressure, reducing the amount of spills and fills during compilation.

Comparing the timing results of the naïve approach to providing fake side-channel leakage with the chaff one we note that the latter is roughly twice as slow as the former one. Such a slowdown is to be ascribed to the overhead imposed by the control managing instructions, and is a constant factor regardless of the number of employed chaff keys.

Table 2 reports the code sizes provided, computed directly from the binaries to be uploaded on the platform, considering the entire `.text` section, which is stored on the Flash memory of the target μC. All the ciphers did not undergo a loop unrolling transformation for the main round-repeating loop: this in turn led to a small performance penalty with respect to the performance results provided in [9], although yielding a significant code size reduction. The code sizes reported concern only the chaff-based implementations as the overhead over the baseline caused by the outer loop iterating over the keys is negligible. We note that the best performers in terms of code size,

Table 2: Code size overheads for the examined set of block ciphers. Dashed-out cells indicate that the resulting binary cannot be fit within the device Flash memory. Gray cells show the best performer for a given number of chaff keys

	Code Size [kiB]						
c+1	1	2	4	8	16	32	64
AES–T	2.46	5.0	9.7	28.5	126.5	596.6	–
Clefia	6.26	14.3	29.1	87.0	407.5	–	–
Present	2.73	4.4	8.5	22.0	71.5	273.8	–
Simon	2.49	3.4	4.5	10.6	31.3	144.9	648.5
Speck	2.11	2.7	3.7	6.4	17.7	57.9	303.0
XTEA	1.74	2.1	3.0	5.3	17.7	67.3	283.5

namely Speck 128/128 and XTEA fit into less than 10% of the target Flash memory when employing up to 31 chaff keys. On the other hand, the spill prone nature of the other cipher implementations raises their code size significantly, as a non-negligible number of `load-store` operations are added to the binary. This, in turn, causes AES, Clefia, and Present not to be usable with all the examined number of chaff keys. An interesting consideration is the fact that lightweight ciphers such as XTEA, even when protected with the chaff countermeasure, still retain a code size below the one of the unprotected Clefia (namely, up to c+1=8) and AES (up to c+1=2). This can be seen as an opportunity to exploit the performance and code size advantage provided by a lightweight cipher to insert a reactive countermeasure, while keeping its code size below the alternative provided by an unprotected non-lightweight one.

6. Conclusion

In this work we described a reactive countermeasure against side-channel attacks, by means of the addition of chaff computations. We pointed out the scenario where such a countermeasure can be employed and reported a detailed security analysis. We have shown how the actual security margin provided by the countermeasure grows linearly in the amount of chaff keys employed, whenever a cascade side-channel attack strategy is employed, as opposed to a straightforward one. We described an automated application strategy for the chaff countermeasure as a set of LLVM compiler passes, and analyzed it on a platform constituted by software block cipher implementations on a Cortex-M4 microcontroller. The results report chaff-protected implementations with 7 to 15 chaff keys, while keeping sub-ms running times and the code size smaller than 20 kiB.

Acknowledgements

This work was supported in part by the EU grants for the actions: “SafeCOP” (ECSEL RIA) Grant agreement no. 692529, and “M2DC” (H2020 RIA) Grant agreement no. 688201.

References

[1] J. L. Galea, E. D. Mulder, D. Page, M. Tunstall, SoC It to EM: ElectroMagnetic Side-Channel Attacks on a Complex System-on-Chip, in: Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th Int.l Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings, Vol. 9293 of LNCS, Springer, 2015, pp. 620–640.

[2] C. H. Gebotys, B. A. White, A Sliding Window Phase-Only Correlation Method for Side-Channel Alignment in a Smartphone, ACM Trans. Embedded Comput. Syst. 14 (4) (2015) 80.

[3] P. C. Kocher, J. Jaffe, B. Jun, P. Rohatgi, Introduction to Differential Power Analysis, J. Cryptographic Engineering 1 (1) (2011) 5–27.

[4] J. Heyszl, S. Mangard, B. Heinz, F. Stumpf, G. Sigl, Localized electromagnetic analysis of cryptographic implementations, in: Topics in Cryptology - CT-RSA 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings [23], pp. 231–244.

[5] G. Agosta, A. Barenghi, M. Maggi, G. Pelosi, Compiler-based side channel vulnerability analysis and optimized countermeasures application, in: The 50th Annual Design Automation Conference 2013, DAC’13, Austin, TX, USA, May 29 - June 07, 2013, ACM, 2013, pp. 81:1–81:6.

[6] O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, I. Verbauwhede, Consolidating Masking Schemes, in: Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, CA, USA, August 16-20, 2015, Proceedings, Part I, Vol. 9215 of LNCS, Springer, 2015, pp. 764–783.

[7] G. Agosta, A. Barenghi, G. Pelosi, A code morphing methodology to automate power analysis countermeasures, in: The 49th Annual Design Automation Conference 2012, DAC ’12, San Francisco, CA, USA, June 3-7, 2012, ACM, 2012, pp. 77–82.

[8] G. Agosta, A. Barenghi, G. Pelosi, M. Scandale, A multiple equivalent execution trace approach to secure cryptographic embedded software, in: The 51st Annual Design Automation Conference 2014, DAC’14, San Francisco, CA, USA, June 1-5, 2014, ACM, 2014, pp. 210:1–210:6.

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

[10] C. Stoll, Stalking the Wily Hacker, Comm. ACM 31 (5) (1988) 484–497.

[11] B. M. Bowen, S. Hershkop, A. D. Keromytis, S. J. Stolfo, Baiting Inside Attackers Using Decoy Documents, in: Security and Privacy in Communication Networks - 5th Int.l ICST Conference, SecureComm 2009, Athens, Greece, September 14-18, 2009, Revised Selected Papers, Vol. 19 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Springer, 2009, pp. 51–70.

[12] A. Juels, T. Ristenpart, Honey Encryption: Encryption beyond the Brute-Force Barrier, IEEE Security & Privacy 12 (4) (2014) 59–62.

[13] A. Juels, T. Ristenpart, Honey Encryption: Security Beyond the Brute-Force Bound, in: Advances in Cryptology - EUROCRYPT 2014, Copenhagen, Denmark, May 11-15, 2014. Proceedings, Vol. 8441 of LNCS, Springer, 2014, pp. 293–310.

[14] NXP Semiconductors, NXP Keyless Entry/Go solutions, `http://www.nxp.com/documents/leaflet/75017275.pdf` (2015).

[15] Atmel, Embedded AVR Microcontroller Including RF Transmitter and Immobilizer LF Functionality for Remote Keyless Entry, `http://www.atmel.com/Images/Atmel-9182-Car-Access-ATA5795C_Datasheet.pdf` (2014).

[16] D. Moon, K. Hwang, W. Lee, S. Lee, J. Lim, Impossible Differential Cryptanalysis of Reduced Round XTEA and TEA, in: Fast Software Encryption, 9th Int.l Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Vol. 2365 of LNCS, Springer, 2002, pp. 49–60.

[17] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, L. Wingers, The SIMON and SPECK lightweight block ciphers, in: Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015, ACM, 2015, pp. 175:1–175:6.

[18] ISO/IEC Joint Technical Committee 1/SC 27., ISO/IEC 29192-2:2012: Lightweight cryptography–Part 2: Block ciphers (2012).

[19] A. Moradi, M. Kasper, C. Paar, Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures - An Analysis of the Xilinx Virtex-4 and Virtex-5 Bitstream Encryption Mechanism, in: Proc. of CT-RSA 2012 [23], pp. 1–18.

[20] STMicroelectronics, STM32L0 Series - Ultra Low Power μCs, `www.st.com/web/en/catalog/mmc/FM141/SC1169/SS1817` (2015).

[21] Silicon Labs, EFM32 - Energy Friendly μCs Line, `https://www.silabs.com/SiteDocs/selector-guide/mcu/efm32-selector-guide.pdf` (2015).

[22] Microchip, PIC32MX Microcontroller Family, `https://www.microchip.com` (2015).

[23] Topics in Cryptology - CT-RSA 2012, San Francisco, CA, USA, 2012. Proceedings, Vol. 7178 of LNCS, Springer, 2012.