A Code Morphing Methodology to Automate Power Analysis Countermeasures

Giovanni Agosta
Politecnico di Milano
Piazza Leonardo da Vinci, 32
20133 Milano, Italy
agosta@elet.polimi.it

Alessandro Barenghi
Politecnico di Milano
Piazza Leonardo da Vinci, 32
20133 Milano, Italy
barenghi@elet.polimi.it

Gerardo Pelosi
Politecnico di Milano
Piazza Leonardo da Vinci, 32
20133 Milano, Italy
pelosi@elet.polimi.it

ABSTRACT

We introduce a general framework to automate the application of countermeasures against Differential Power Attacks aimed at software implementations of cryptographic primitives. The approach enables the generation of multiple versions of the code, to prevent an attacker from recognizing the exact point in time where the observed operation is executed and how such operation is performed. The strategy increases the effort needed to retrieve the secret key through hindering the formulation of a correct hypothetical consumption to be correlated with the power measurements. The experimental evaluation shows how a DPA attack against OpenSSL AES implementation on an industrial grade ARM-based SoC is hindered with limited performance overhead.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application Based Systems**]: Microprocessor/microcomputer applications;
C.5.3[**Computer System Implementation**]: Microcomputers[portable devices];

General Terms

Security

Keywords

Power Analysis Attacks, Software Countermeasures, Dynamic Code Transformation, Polymorphic Code

1. INTRODUCTION

The general trend in embedded hardware security shows a large use of cryptographic operations, and an increasing attention towards tamper resistant designs and countermeasures against side-channel attacks like power analysis and fault injection. Indeed, it is effectively proven that the physical access to an embedded device may enable the recovery of sensitive information, which is otherwise supposed to be hidden [1,2,8], through exploiting both the implementation weaknesses of the cryptographic operations and specific features provided by the underlying hardware platform. Differential Power Analysis (DPA) introduced in [6] has been proven a powerful threat that triggered a flourishing research branch with a wide range of improvements and countermeasures both in hardware and software. DPA attacks against an unprotected implementation of a cryptographic algorithm follow a common workflow: first of all they measure the power consumption (*power traces*) of the targeted device for a high number of runs (i.e. considering a high number of input/output values). Subsequently, they select an intermediate operation of the algorithm employing a part of the secret key, and compute an expected consumption for every possible value of the key portion, according to a model of the triggered switching activity (e.g. the Hamming weight of the outputs). Finally, the predicted consumption values are matched against each sample of the recorded power traces to assess which key hypotheses fit better 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. The principal countermeasures against power analysis are split into two categories [8]: *masking* and *hiding*. Masking aims to invalidate the link between the predicted hypothetical power consumption values, associated to the selected intermediate operation, and the actual values processed by the device. 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 target algorithm is modified to correctly process each share and to recombine them at the end of the computation. A masking scheme with only two shares is composed by the values v_m and m, where m is a randomly chosen mask and v_m is a share such that the value v to be protected can be derived as $v{=}v_m{\diamond}m$, with $\diamond$ denoting an invertible binary operation. To compensate for this countermeasure, more sophisticated DPA attacks, known as high-order DPAs rely on predicting the consumption of all the operations handling the shares and try to obtain a combination of them independent from the masking values. This value must subsequently be correlated with an analogous combination of the measured consumption values, employing the same techniques of a common (first order) DPA. The technical effort in carrying out an high-order DPA attack quickly grows as the order (i.e. the number of shares) increases just as the time/space resources to be employed in recording a larger number of power traces. It is commonly accepted that a masking scheme with a large number of shares makes DPA attacks either practically unfeasible or inconvenient. Typically, engineering solutions strive to introduce a moderate overhead with respect to the unprotected version of the primitive, resorting to the combination of two-share masking schemes and hiding techniques [9]. Hiding methods aim to conceal the relation between the power consumption and the operations performed by the target algorithm to compute the intermediate values. The protection strategies employed in the open literature, to secure software implementations, are based on execution flow randomization via shuffling the order of some instructions (f.i., permuting the sequence of accesses to lookup tables) and inserting random delays built with dummy operations [9, 10].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC '12 June 3-7, 2012, San Francisco, California, USA.
Copyright 2012 ACM 978-1-4503-1199-1/12/06 ...$10.00.

To minimize the performance overhead, the execution must be interleaved with delays in multiple places, keeping the individual delays as short as possible. In this way, an attacker faces a cumulative and hardly predictable sum of delays between the start (the end, respectively) of the algorithm and the location of the observed intermediate operation in time [4]. These techniques only affect the time dimension of the power consumption but they do not change the power consumption characteristics of the operations performed by the target device. In spite of the limits of the aforementioned techniques, software countermeasures are well suited for general purpose processors where no dedicated security features are built in at design time. In addition to this broad range of applicability, software-based countermeasures represent also a viable mitigation mean to restore security into hardware-protected systems, where the underlying hardware protections have been compromised, without the need for an expensive part replacement.

1.1 Contributions

The novel approach proposed in this work is a software countermeasure framework based on the combination of a cryptographic algorithm implementation with a *polymorphic engine* which dynamically and automatically transforms the binary code to be protected. Thus, we propose innovative contributions to two common practices in the field: (i) static generation of the protected code; (ii) manual and often application-specific generation of the protected code. Our method moves the code generation at run-time, and enables the generation of many different versions of the protected code at the designer's will, preventing any attacker from both recognizing the exact point in time where the observed operation is executed and understanding how such operation is actually performed. This methodology separates the creative work of identifying replacements for assembly code snippets from the tedious (but amenable to automatization) work of applying such replacements to the entire code. This strategy largely increases the effort needed to predict the value of a sensitive intermediate result of the considered algorithm and hinders the formulation of a correct hypothetical consumption to be correlated with the power measurements. Polymorphic engines are the key component to build a special class of programs: the ones characterized by the ability to modify parts of their own code. In particular, a polymorphic code is composed of two parts: the polymorphic engine, which never changes, and the target code to be modified. Self-modifying code is used in several areas to provide either optimization or obfuscation: dynamic compilers, and especially fragment linking [5], tamper resistant software and protection against reverse engineering of executable code [7]. We adapt self-modification principles to both swap parts of the target algorithm with different, but semantically equivalent, replacements and to implement concepts such as *masking* and *hiding*. In particular, concerning the hiding countermeasure techniques, our approach provides both time-dimension and switching activity hiding, through changing both the type of operation and the time needed to compute the same intermediate value. With respect to state of the art, we provide: (i) a *generalization*, through allowing several types of countermeasures to be applied in a unifying framework; (ii) an *extension*, through providing variants to existing countermeasures; (iii) an increased *variability* of the protected code, through re-generating it as often as needed by means of dynamic code morphing. To this end, we allow the countermeasure designer to specify, for each operation or group of operations, a set of code transformation templates that can be automatically applied to the binary code of the target cryptographic primitive. Sufficient generality is provided to allow the expression of random delays such as those proposed in [4], as well as to replicate other hiding strategies, such as those shown in [3]. The availability of a wide range of transformations in our framework allows the trade-off between performance overheads and security margin to be finely tuned at design time.

1.2 Case Study

We chose as a case study platform an ARM926-based STMicroelectronics SPEAr SoC, as a representative of a large class of high-end embedded devices where commonly no hardware protection against side channel attacks are employed. The chosen cryptographic primitive is the AES, as implemented in the widely diffused OpenSSL toolkit. The choice of this testbench was driven by the large adoption of this algorithm as a mean to provide data confidentiality. The most common intermediate values employed during an attack to an AES implementation are represented by output of a *load* operation from the S-Box, triggered by the SUBBYTE step and by the output of the post-ADDROUNDKEY state. However, in the selected platform, the consumption model relying on the latter is by far more effective than the one relying on the S-Box, due to the unpredictable power saving on the *load* operations caused by the use of data caches (see Section 4). Even if the number of successful DPA attacks against software implementations on complex SoCs is rather low due to the complexity of such devices, we were able to extract the full AES key from the testbed platform with a sensibly low number of measurements. Once the attack has been proven feasible, we employ the proposed framework to effectively and efficiently counteract the identified vulnerability.

The remainder of the paper is organized as follows. Section 2 provides the definitions necessary to formalize the code morphing operations. Section 3 introduces our code transformation framework, and describes the proposed polymorphic engine. Section 4 presents the experimental evaluation on the target case study. Section 5 provides a brief overview of closely related works. Section 6 draws some conclusions and highlights future directions.

2. SEMANTIC EQUIVALENCE OF CODE FRAGMENTS

Our approach of building a different version of a static binary code at run-time prior to executing it relies on the substitution of each static code fragment with one of its randomly-chosen variants, preserving the black-box behavior of the original static code. The notions of *code fragment* and *semantic-equivalence* between code fragments are formally defined as follows.

DEFINITION 2.1 (CODE FRAGMENT). *A sequence of instructions* $\mathcal{I}$=($\mathtt{inst}_1$,...,$\mathtt{inst}_s$), *$s \geq 1$, is a code fragment when each term* $\mathtt{inst}_j \in \mathcal{I}$ *is executed exactly once,* $\mathtt{inst}_j$ *executes before* $\mathtt{inst}_{j'}$ $\forall\, j, j'$ *such that* $j<j' \leq s$, *and no other instruction is executed between* $\mathtt{inst}_j$ *and* $\mathtt{inst}_{j+1}$, $1 \leq j < s$.

Intuitively, the sequence of terms composing a code fragment must not include any branch or privileged instruction like a supervisor call, an I/O or an I/O MMU-bypass operation.

DEFINITION 2.2 (SEMANTIC EQUIVALENCE). *Let* $\mathcal{I}$, $\widetilde{\mathcal{I}}$ *be two code fragments, and let* $\mathcal{A}(\mathcal{I})$, $\mathcal{A}(\widetilde{\mathcal{I}})$ *be the sets of* live-out *variables related to* $\mathcal{I}$ *and* $\widetilde{\mathcal{I}}$, *respectively (i.e.: registers and memory locations that from the exit of the code fragment on are read at least once). A semantic-preserving relation between* $\mathcal{I}$ *and* $\widetilde{\mathcal{I}}$ *is an equivalence relation,* $\overset{S}{\sim}$, *where* $\mathcal{A}(\mathcal{I})=\mathcal{A}(\widetilde{\mathcal{I}})$ *and the corresponding written live-out values are the same.*

Given a generic code fragment, the problem of constructing a set of semantically equivalent variants is not practically interesting without a precise characterization of the goals to be achieved (f.i. it is easy to generate an infinite set of equivalent code fragments from

a single-instruction loop). Therefore, in the following we formulate sufficient criteria to either map a given code fragment to a semantically-equivalent one or verify if two code fragments have the same semantics. The translation of the original static code is performed through locally scoped substitutions that are easier and more efficient to implement than global ones. Thus, the semantic equivalence of the resulting program is obtained from the composition of sematically equivalent independent code fragments, thus limiting the performance impact of the dynamic translation.

PROPOSITION 2.1 (CODE FRAGMENT EQUIVALENCE). *Let* $\mathcal{I}$=($\mathtt{inst}_1$,...,$\mathtt{inst}_s$), $s \geq 1$, *and* $\widetilde{\mathcal{I}}$=($\mathtt{inst}_1$,...,$\mathtt{inst}_{\tilde{s}}$), $\tilde{s} \geq 1$, *be two code fragments. The semantic equivalence* $\widetilde{\mathcal{I}} \overset{S}{\sim} \mathcal{I}$ *is preserved if: (1) every register,* $\mathtt{reg}_k$, $k \geq 0$, *of the CPU is employed in* $\mathcal{I}$, *and* $\widetilde{\mathcal{I}}$ *according to the following constraints: (1.a)* $\mathtt{reg}_k$ *is either not included in any register assignment operations of* $\mathcal{I}$ *nor* $\widetilde{\mathcal{I}}$ *or (1.b)* $\mathtt{reg}_k$ *is used only in* $\widetilde{\mathcal{I}}$, *where it is spilled to the memory before any assignment and filled back as last action, or (1.c) the collections of possible register values at the end of* $\mathcal{I}$ *and* $\widetilde{\mathcal{I}}$ *(computed with the same initial values) must be the same; (2) the memory assignments derived from the sequence of write-operations in* $\mathcal{I}$ *are preserved in* $\widetilde{\mathcal{I}}$, *with the same values; (3) any memory assignment performed in* $\widetilde{\mathcal{I}}$ *but not in* $\mathcal{I}$ *writes the stack segment in memory locations outside* $\mathcal{A}(\widetilde{\mathcal{I}})$, *i.e.: in memory locations after the position of the stack pointer at the end of* $\mathcal{I}$.

PROOF. The first condition implies that for all CPU registers the corresponding values at the end of $\mathcal{I}$, and $\widetilde{\mathcal{I}}$ are exactly the same. Therefore, the condition required by Definition 2.2 that all *live-out* registers, $\mathcal{A}(\mathcal{I})$ and $\mathcal{A}(\widetilde{\mathcal{I}})$, have the same corresponding values is trivially satisfied. The second condition ensures that memory locations referred in $\mathcal{I}$ and $\widetilde{\mathcal{I}}$ are also written in the same order and with the same values. The last condition allows additional memory assignements in $\widetilde{\mathcal{I}}$ to target a larger portion of the stack segment w.r.t. $\mathcal{I}$, nevertheless restoring the same value of the stack pointer in $\mathcal{I}$ at the end of the fragment $\widetilde{\mathcal{I}}$ as required by the first condition. The memory assignments are guaranteed not to be live-out, which matches the requirement stated in Definition 2.2. □

It is important to note that the sufficient conditions stated in Proposition 2.1 are locally verifiable, while Definition 2.2 employs the liveness property, which must be computed for all memory locations and registers. This requires a global analysis, unfeasible at run-time, and not always feasible at all even at compile-time if the whole address space is considered. By contrast, the conditions in Proposition 2.1 only rely on information which can be retrieved through a linear scan.

3. TRANSFORMATION FRAMEWORK

The proposed framework takes as input a target cryptographic algorithm, and statically compiles it to produce a binary code with proper calls to a run-time library which implements the code morphing engine and is in charge of modifying the target code on the underlying architecture. Figure 1 reports an high level description of the code transformation flow. The static compiler operates within the standard compilation process from source code to machine code. The source program is written as a `C` code with some specific *annotations*. Code annotations are usually employed to encode more information for the compiler than the explicit source code, such as hints about how to organize or optimize the intermediate representations of the code. In our case, we employ a custom gcc `__attribute__` (defined as: `__attribute__((secure))` before a function or variable declaration) to specify to the compiler

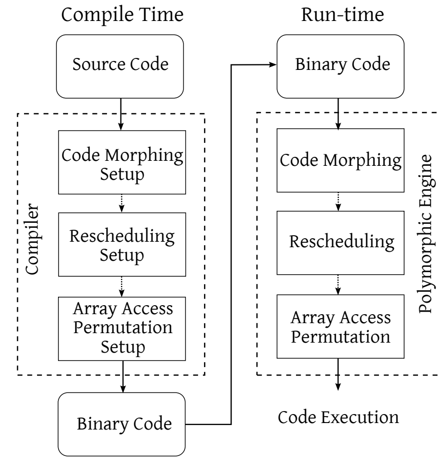


Figure 1: Compilation and run-time code transformation flow

which functions and data structures need to be secured. In the case of functions, for each stage of the static compiler (left side of Figure 1), the transformation flow wraps their calls with invocations of the *Code Morphing*, *Rescheduling*, and *Array Access Permutation* run-time routines. The case of array variable declarations is managed by the third stage of the static compiler (*Array Access Permutation Setup*) which makes the access to every array cell as an indirect access through using a further array (with the same length) containing a permutation of indexes. The *Code Morphing Setup* stage also allocates in memory the data structures (*tile set*) containing the knowledge base needed to perform the code morphing, through substituting each code fragment with one of its semantically equivalent variants included in the *tile set*. At run-time, the polymorphic engine, reported on the right-hand side of Figure 1, changes the original binary code through performing three steps: *Code Morphing*, *Rescheduling*, and *Array Access Permutation*. The Code Morphing stage, which is the core of the proposed approach, is described in greater detail in the next section.

The *Rescheduling* step adds a level of obfuscation with respect to the possible recognition (or classification) of the executed code through rearranging the instructions within a finite window, in such a way to preserve the data dependencies. This operation is performed by means of a single scan of the code, from the bottom to the top. At each step, a single instruction $\mathtt{inst}_i$ and a window of k instructions preceding it $\{\mathtt{inst}_{i-1},\ldots,\mathtt{inst}_{i-k}\}$ are considered. The dependencies of $\mathtt{inst}_i$ are computed, and the earliest position $i-h$ ($h \leq k$) which it can take is determined. Then, a random position $i-l$ in the range $[i-h, i]$ is selected, and the instructions are reordered consequently. Finally, the next value of i is set to $i-l$, and the process continues until the start of the code is reached. This technique is based on the well-known code scheduling theory employed in the field of compiler optimization. The code scheduling theory provides a set of semantic preserving schedules for a given code. Where the compiler practices select a schedule to minimize latency and/or power consumption, our goal is to randomly change a given schedule with a different (but equivalent) one to alter the shape and mutual alignment of the power traces.

The *Array Access Permutation* step applies a random permutation to the allocated array indexes, hiding the access patterns to the substitution table of a symmetric cipher. The access pattern hiding technique has been proposed and detailed in [9, 10]. Since the access to substitution tables is not the preferred attack point in our testbed platform, for the sake of brevity, we refer to the aforementioned works for further details.

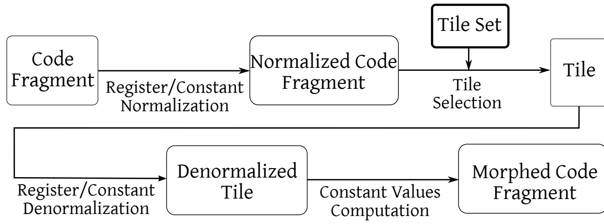


Figure 2: Code morphing engine workflow

3.1 Code Morphing Engine

To apply code morphing to a wide variety of code fragments, it is necessary to represent them in a way that abstracts from the actual registers and immediate values employed (e.g., `add r2,r5,r2` only differs from `add r4,r11,r4` in the registers used, and both can be abstracted to a normal form `add r0,r1,r0`). To clarify the normalization process, we will now formally define the concepts of *register normalization* and *constant normalization*.

DEFINITION 3.1 (REGISTER NORMALIZATION). *Given a code fragment $\mathcal{I}$=(`inst`$_1$,...,`inst`$_s$), $s\geq 1$, where `inst`$_i$, with $0\leq i\leq s$, is a data processing assembly instruction specified as* `opCode` *`dest`, `src`, `operand`, a* register normalization *is a map of the register names appearing in the fields {`dest`, `src`, `operand`} of the instruction sequence, into register names starting from* `r0` *for the first register in* `inst`$_1$ *on.*

DEFINITION 3.2 (CONSTANT NORMALIZATION). *Given a code fragment $\mathcal{I}$=(`inst`$_1$,...,`inst`$_s$), $s\geq 1$, where `inst`$_i$, with $0\leq i\leq s$, is a data processing assembly instruction specified as* `opCode` *`dest`, `src`, `operand`, a* constant normalization *is a map of the immediate values appearing in the fields {`src`, `operand`} of each instruction, into immediate values starting from* `#0` *on, to be interpreted as symbolic constants in the resulting instruction.*

Building on the two previous definitions the *normalized code fragment* can be defined as follows:

DEFINITION 3.3 (NORMALIZED CODE FRAGMENT). *Given a code fragment $\mathcal{I}$=(`inst`$_1$,...,`inst`$_s$), $s\geq 1$, a* normalized code fragment, *$\overline{\mathcal{I}}$, is the sequence of instructions resulting from the application of both register (Definition 3.1) and constant (Definition 3.2) normalization mappings.*

In principle, for each normalized code fragment $\overline{\mathcal{I}}_i$, $i\geq 1$, a set of $m\geq 1$ semantically equivalent fragments $\mathbf{S}_{\overline{\mathcal{I}}_i}=\{\tilde{\overline{\mathcal{I}}}_{i,0},\ldots,\tilde{\overline{\mathcal{I}}}_{i,m-1}\}$, $\tilde{\overline{\mathcal{I}}}_{i,j}\overset{S}{\sim}\overline{\mathcal{I}}_i, \forall j\in\{0,\ldots,m-1\}$, can be written through applying the sufficient conditions specified by Proposition 2.1. Note that any non-trivial $\mathbf{S}_{\overline{\mathcal{I}}_i}$ set must be created manually.

EXAMPLE 3.1. *Consider a code fragment $\mathcal{I}$ composed by a single instruction* `inst`$_1$*:* `eor r5,r0,r4`*, which writes into* `r5` *the bitwise exclusive-or of the values in* `r0` *and* `r4` *(*`r5`$\leftarrow$`r0`$\oplus$`r4`*). Its normalized form is computed as* `eor r0,r1,r2`*, while a corresponding semantically equivalent fragment is given by $\tilde{\overline{\mathcal{I}}}$=(* `bic r0,r1,r2;bic r3,r2,r1;orr r0,r0,r3` *)$\overset{S}{\sim}\overline{\mathcal{I}}$, where the ARM ISA instruction* `bic r0,r1,r2` *computes* `r0`$\leftarrow$`r1`$\wedge\neg$`r2`*. The use of extra registers, as* `r3`*, must be managed through* clobbering *the temporary registers before their use, and restoring them at the end of the instruction sequence in $\overline{\mathcal{I}}$.*

It is possible to generate large $\mathbf{S}_{\overline{\mathcal{I}}_i}$ sets, which must be stored in memory and used as a knowledge base for the Code Morphing phase. Therefore, a key issue is to provide a compact representation for them. Note that, for the same $\mathcal{I}$, we can generate several $\overline{\mathcal{I}}_i$ which differ only by the values assumed by some of the constants involved.

Given the normalized fragment $\overline{\mathcal{I}}$=(`and r0 r1 #0`), the designer may want to replace it by applying the following transformation:

$$\texttt{r0} \leftarrow (\texttt{r1} \wedge (\texttt{\#0} \oplus \texttt{const1})) \diamond (\texttt{r1} \wedge (\texttt{\#0} \oplus \texttt{const2}))$$

where $\diamond$ is $\wedge$ or $\vee$ and `const1` and `const2` are additional symbolic constants such that `const1`$\oplus$`const2`=`0xf...f` if $\diamond$ is $\wedge$ and `const1`$\oplus$`const2`=`0x0` if $\diamond$ is $\vee$. Two semantically equivalent fragments $\tilde{\overline{\mathcal{I}}}_0$, $\tilde{\overline{\mathcal{I}}}_1$, to $\overline{\mathcal{I}}$ are:

$\tilde{\overline{\mathcal{I}}}_0$=(	$\tilde{\overline{\mathcal{I}}}_1$=(
`and r0,r1,#0`$\oplus$`const1`	`and r0,r1,#0`$\oplus$`const1`
`and r2,r1,#0`$\oplus\neg$`const1`	`and r2,r1,#0`$\oplus$`const1`
`and r0,r0,r2`	`orr r0,r0,r2`
)	)

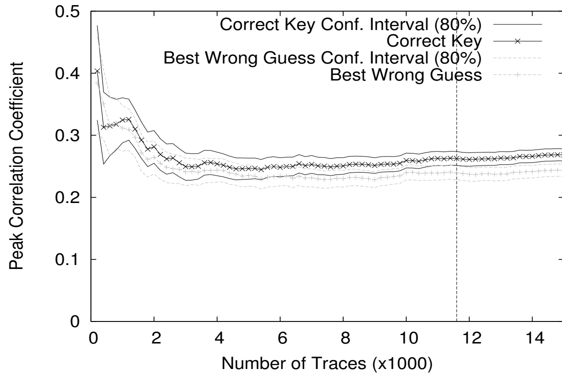
where `const1` is an additional symbolic constant that can assume any value, whereas the symbolic constants #0, #1,... derived from the constant normalization are constrained to their original immediate value, and `r2` is a clobbered register. We formalize the concept of a normalized code fragment augmented with operations on symbolic constants as follows.

DEFINITION 3.4 (TILE). *Given a normalized code fragment $\overline{\mathcal{I}}_i$, a tile t_i is a set of normalized fragments semantically equivalent to $\overline{\mathcal{I}}_i$, distinguished only by the values of additional symbolic constants* `const_j`*. These constants appear as immediate operands in the instructions of the code fragments, either alone or as part of constant expressions containing arithmetic-logic operators and symbolic constants from $\overline{\mathcal{I}}_i$.*

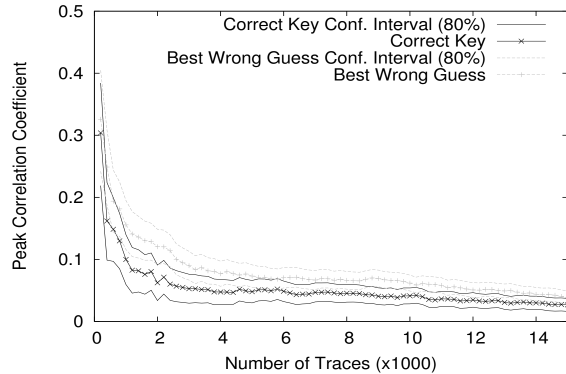
The expressions on symbolic constants are encoded within the bits used to encode the immediate operand fields of each instruction in the tile. As for the code fragments, it is possible to write a set S_{t_i}, which is a collection of tiles for the normalized code fragment $\overline{\mathcal{I}}_i$. For each normalized code fragment $\overline{\mathcal{I}}_0, \overline{\mathcal{I}}_1, \ldots$, the collection of the semantically equivalent tiles $\{\mathbf{S}_{t_0}, \mathbf{S}_{t_1}, \ldots\}$, which must be protected, represents the whole *tile set* of the morphing engine. The complete workflow of the code morphing engine, shown in Figure 2, is composed as follows. First, the input code fragment $\mathcal{I}$ is mapped to a normalized code fragment $\overline{\mathcal{I}}$ through the Register/Constant Normalization step. Then, a tile is randomly selected from the Tile Set to replace the normalized code fragment. The Register/Constant Denormalization procedure is then applied to the registers and symbolic constants. Finally, in the Constant Values Computation step, any symbolic constant is replaced by a random value, and the constant expressions encoded in the tile are evaluated to obtain the immediate operands.

4. EXPERIMENTAL EVALUATION

The experimental platform used to provide a validation of our framework is an ARM-based STMicroelectronics SPEAr Head200 development board. The SPEAr SoC is based on a 32-bit ARM926EJ-S processor running at 133 MHz, without any OS, for the sake of more precise analysis of the results. The AES binary, based on OpenSSL 1.0.0d, with 4 T-tables, is run directly from the U-Boot bootloader. The attack exploits the outcome of the *xor* operation in the first ADDROUNDKEY, which is stored in a register.



(a) Unprotected system results



(b) Protected system results

Figure 3: Highest correlation values for correct and best-wrong key guesses as a function of the number of traces, together with their 80% confidence intervals. For the unprotected system, at least 11600 traces are needed to distinguish two separated confidence intervals, i.e. the correct key guess (a). In the case of the protected system (one morph action every 100 traces), all the correlation peaks have statistically negligible values, regardless of the key hypothesis and the increase in the number of traces (b)

4.1 Performance evaluation

Both the protected and the unprotected systems have been evaluated to assess the time overhead introduced by the countermeasure. Through employing a GCC 4.3 based cross-compile toolchain, we observe that the original AES algorithm is composed of 536 instructions, clustered in 20 distinct normalized instructions. We employ three, 4-instruction long, tiles to protect the 64 `eor` code fragments in the whole AES. Thus, instruction replacement adds on average 4 computational instructions for each substituted `eor`, plus a load/store pair to handle *clobbering*. Still, we expect a reduced overhead, as most of the AES execution time is spent in accessing memory for the T-table lookups. The timings have been gathered directly via trace length measurements on the oscilloscope. On average, a run of the unprotected AES algorithm takes 228.8 μs, while a run of the protected one runs for 245 μs, thus resulting in a performance hit of 8.2% of the AES execution time. The overhead due to the code morphing amounts to 90 ms per morphing action. Since the code morphing algorithm is intrinsically memory bound, the majority of this delay is ascribed to accesses to the off-chip memory in our platform. To cope with the additional performance hit, the number of calls to the code morpher over the encryption algorithm runs can be tuned to bring the amortized encryption time within acceptable bounds for the target system. The following section shows how the overhead is reduced to a fraction of the time of one encryption run, and evaluates the security margin provided.

4.2 Differential power analysis

Measured power traces were obtained with an Agilent *Infinium*DSO80204B oscilloscope and an active Agilent 1131A differential voltage probe with a 3.5 GHz analog bandwidth.The oscilloscope features 4 independent analog channels, a 2 GHz analog bandwidth, coupled with an 8-bit ADC capable of recording 40 Gsample/s, with a noise floor of 3 mV RMS, and a minimum vertical resolution of 10 mV. The measured power traces have been acquired using a sampling frequency of 500 Msamples/s over an acquisition window of 100 ksamples. The trigger signal is provided via a GPIO pin on the board and collected via a passive probe connected via an Agilent E2697A impedance adapter.

The power measurements have been obtained via measuring the voltage drop at the ends of a 1 Ω SMD resistor inserted on the SPEAr SoC power supply line. To reduce measurement noise, each trace is the result of the average of 32 measurements with the same plaintext and code. This represents a worst case scenario, as a real world attacker will not be able to choose which code variant is running to get averaged measurements. An attacker might trade off measurements of different plaintexts to gain noise reduction via averaging; however, the maximum number of measurements with a single code variant is bound by a design parameter, i.e. the number of encryption runs before a call to the morphing engine is made. A first order Differential Power Analysis against both unprotected and protected AES implementations has been used as testbench. The employed consumption model is the Hamming weight of one byte of the output of the first ADDROUNDKEY operation. This operation is computed 32 bits at a time, since that is the size of the ARM architecture word length. The analysis computes the sample estimation r of Pearson's correlation coefficient ρ between each sample of the actual power consumption measurements (traces) of the device and the consumption model for each possible hypothetical value of the involved key part. Subsequently, the maximum values of r obtained for each key hypothesis are sorted in decreasing order. For the attack to succeed, the confidence interval I_r of the maximum value should not overlap with any of the others. The correct key hypothesis can be successfully obtained when enough traces are gathered, as the width of I_r decreases when the number of measurements increases. An unbiased estimator for the Pearson correlation coefficient ρ is: $\hat{\rho}{=}r\left(1+\frac{1-r^2}{2(n-3)}\right)$, where n is the number of employed samples. To obtain the boundaries of the interval I_r, the probability $\text{Prob}\{\hat{\rho}\in[r_l,\, r_u]\}{\geq}\gamma$ is evaluated for a chosen confidence level γ. The theoretical correlation coefficient for the correct key hypothesis ρ_c is 0.250 since the observed operation is performed on 32 bits at a time, while the consumption model takes into account only 8 of them. By contrast, the key hypotheses differing by a single bit from the correct one (the best wrong guessed) has a theoretical correlation coefficient $\rho_w{=}0.218$. The values of the estimators $\hat{\rho}_c, \hat{\rho}_w$ will converge to ρ_c and ρ_w.

Figure 3a shows that 11600 traces are necessary to distinguish the correct key hypothesis from the best wrong guess with a confidence level $\gamma{=}0.8$. Thus the architecture does not provide any embedded protection against side channel attacks, despite the complexity of the SoC.

Two crucial factors for a power analysis success are represented by the knowledge of the implementation strategy of the attacked operation, which allows to infer its consumption model, and the perfect time alignment of each trace. The devised countermeasure

Table 1: Impact of the number of runs among morphing actions on both the security margin of the system (i.e. overlap of the correct key and best wrong guess confidence intervals) and execution time of a single protected AES encryption (optimal tradeoffs in grey). Execution time of a plain AES is 228.8 μs

Code Morphing Interval [no. of runs]	Confidence Intervals Overlap [%]	Average Execution Time
100	79.55	×5.00
200	79.32	×3.04
400	79.03	×2.05
600	78.89	×1.73
800	78.89	×1.56
1000	78.98	×1.46
2000	79.76	×1.27
3000	79.04	×1.20
4000	75.16	×1.17
5000	67.64	×1.15
6000	56.94	×1.14
11600	0.00	×1.10

operates on both factors, thus actively hindering the attack. Consequentially, it is sufficient to perform a code morphing action often enough to avoid the collection of a significant number of traces by the attacker. The maximum number of measurements with the same (albeit unknown) code variant Δn that an attacker is able to collect is thus a design-time-chosen parameter indicating the security margin of the system.

Figure 3b shows the result of the attack performed while our protection methodology was in action with Δn=100. In this case, the confidence interval for the correct key hypothesis and the best wrong guess never separate. In addition to this, the correct key has a sample correlation coefficient lower than the best wrong guess. As a further validation of our approach, we correlated the key hypotheses evaluated through the same consumption model with random values, obtaining a peak sample correlation value higher ($\sim$0.08) than both the previous estimates for $\hat{\rho}_w$ and $\hat{\rho}_c$. Thus, collecting a greater number of traces will not be useful due to the negligible values of the obtained sample correlation estimates.

A practical measure of the security margin is given by the overlap percentage of the confidence intervals of the correct key and the best wrong guess. We note that this measure is a conservative gauge of the actual security margin, as the attacker fails to retrieve the key also when the confidence interval of the best wrong guess is both disjoint and higher than the one of the correct key. In this case, the overlap percentage is zero but the attack does not succeed. However, the opposite scenario does never happen as an overlap of the confidence intervals unquestionably indicates the indistinguishability of the corresponding key hypotheses. The security margin obtained via code morphing for the target platform can be traded off to achieve a lower computational overhead per encryption run. The computational overhead for the encryption is composed of a fixed cost determined by the tile substitution action and an amortized cost over Δn runs due to the call to the morphing engine. Table 1 reports the trends of the security margin (i.e. the confidence intervals overlap) and the average execution time of single AES as a function of the value of Δn. We verified that the security margin of this platform does not report significant hits up to an attack with a half of the traces needed to retrieve the correct key on an unprotected implementation. The optimal trade-off points for this platform are represented by running the code morpher once every 2000–3000 AES runs, as this parameter choice preserves a high security margin, while obtaining an acceptable (around 20%) performance overhead. Raising further the number of AES runs per morphing action drastically reduces the security margin of the system, without a significant performance improvement.

5. RELATED WORK

The distinguishing feature of our solution lies in the code substitution technique, but schemes such as those proposed in [4, 8–10] can be implemented within our framework by means of specific tiles. In these works, the results regarding the described implementations are mostly related to microcontroller platforms and exhibit case-study specific execution times ranging from two to more than fifty times the baseline. When considering attacks based on an a-posteriori model [8], such as *template* attacks, the very high number of code variants provided by our countermeasure would require a prohibitive number of traces to obtain a reliable model, since a significant quantity of information should be gathered for all of them. In [3] the authors propose a framework that employs an information theoretic metric to identify the most sensitive instructions of a software implementation of AES on an 8-bit microcontroller and apply a static local code modification implementing *random precharging*. W.r.t. [3], we propose an automated, dynamic code morphing approach, which can produce a much larger number of different semantically equivalent code versions, and it is also able to apply hiding and masking techniques together in a unified framework.

6. CONCLUDING REMARKS

We presented a framework to automate the application of DPA software countermeasures at run-time and described a code morphing toolchain that proved to be efficient while ensuring protection for any cryptographic primitive. The proposed approach can be applied to either the whole algorithm or to the subset of vulnerable instructions to enhance performances. To our knowledge this is the first work providing this level of protection while being practically viable. The analyzed case study showed how to counter DPA attacks with an acceptable performance overhead. The overhead may be further reduced when protecting a multi-core platform via concurrently executing the encryption routine and the morphing action. Future works will target microcontrollers where the code is in a read-only memory via morphing the execution flow with a series of random jumps along a database of code fragments

7. REFERENCES

[1] A. Barenghi, G. Pelosi, and Y. Teglia. Improving first order differential power attacks through digital signal processing. In *Proc. of the 3rd Int'l Conf. on Security of Information and Networks*, SIN'10, pages 124–133. ACM, 2010.

[2] A. Barenghi, G. Pelosi, and Y. Teglia. Information leakage discovery techniques to enhance secure chip design. In *Proc. of the 5th IFIP WG 11.2 Int'l Conf. on Information Security Theory and Practice: security and privacy of mobile devices in wireless communication*, WISTP'11, pages 128–143. Springer-Verlag, 2011.

[3] A. G. Bayrak, F. Regazzoni, P. Brisk, F.-X. Standaert, and P. Ienne. A first step towards automatic application of power analysis countermeasures. In *Proc. of the 48th Design Automation Conference*, DAC'11, pages 230–235. ACM, 2011.

[4] J.-S. Coron and I. Kizhvatov. Analysis and improvement of the random delay countermeasure of ches 2009. In *Proc. of the 12th Int'l Workshop on Cryptographic Hardware and Embedded Systems*, CHES'10, pages 95–109. Springer-Verlag, 2010.

[5] E. Duesterwald. Dynamic compilation. In *The Compiler Design Handbook*, pages 739–762. CRC Press, Inc., 2002.

[6] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology*, CRYPTO'99, pages 388–397. Springer-Verlag, 1999.

[7] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. of the 10th ACM Conf. on Computer and Communications Security*, CCS'03, pages 290–299. ACM, 2003.

[8] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer-Verlag, 2007.

[9] M. Rivain, E. Prouff, and J. Doget. Higher-order masking and shuffling for software implementations of block ciphers. In *Proc. of the 11th Int'l Workshop on Cryptographic Hardware and Embedded Systems*, CHES'09, pages 171–188. Springer-Verlag, 2009.

[10] S. Tillich and C. Herbst. Attacking state-of-the-art software countermeasures–a case study for aes. In *Proc. of the 10th Int'l Workshop on Cryptographic Hardware and Embedded Systems*, CHES'08, pages 228–243. Springer-Verlag, 2008.