IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 2, 2021, accepted May 1, 2021, date of publication May 6, 2021, date of current version May 14, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3077977

Metis: An Integrated Morphing Engine CPU to Protect Against Side Channel Attacks

FRANCESCO ANTOGNAZZA, ALESSANDRO BARENGHI, AND GERARDO PELOSI, (Member, IEEE)
Department of Electronics, Information and Bioengineering, Politecnico di Milano, 20133 Milan, Italy
Corresponding author: Gerardo Pelosi (gerardo.pelosi@polimi.it)

ABSTRACT Power consumption and electromagnetic emissions analyses are well established attack avenues for secret values extraction in a large range of embedded devices. Countermeasures against these attacks are approached at different levels, from modified logic styles, to changes in the software implementations. In this work, we propose a microarchitectural modification to a compact RISC-V SoC, the OpenTitan open source silicon root of trust, providing a *code morphing* countermeasure against power and electromagnetic emissions side channel attacks. Our approach allows the countermeasure to be applied transparently, without the need for any software modification to the cryptographic primitive running on OpenTitan. Our microarchitecture integration of a morphing engine also allows us to provide transparent protection to memory operations. We validate our approach through measurements on an actual FPGA prototype on a Xilinx Artix-7. Our integrated morphing engine increases the FPGA resource consumption by less than 8%, plus the resources required by an RNG of choice, with respect to the original OpenTitan SoC. Our design shows a side channel attack resistance improvement of at least 250× in the Measurements-To-Disclose metric with respect to the unprotected design. We benchmark the performance of our proposed architecture on all the ISO/IEC standard symmetric block ciphers, including, among the other AES, reducing the execution time overhead by 21× to 141× with respect to a continuously morphing software solution.

INDEX TERMS Applied cryptography, code morphing, computer security, side channel attacks, power consumption attack countermeasures.

I. INTRODUCTION

Side channel attacks have long been proven to be a concrete threat against the security of computing systems. Indeed, their ability to rely on the accidental transmission of information via environmental parameters of the working regime of a computing device allows them to retrieve confidential information from fully functional cryptographic implementations. Since the work of Kocher [37], where the power consumption of a smartcard was employed to derive the secret key of the DES cipher running on it, side channel attacks were used to breach the security of a large variety of devices, ranging from inexpensive microcontrollers for IoT devices and RFIDs [21], [46], [50] through mid-range system on chips [7], [9] to full desktop and laptop grade CPUs [24], [25], [36], [39], [45].

The associate editor coordinating the review of this manuscript and approving it for publication was Ilsun You.

Side channel attacks rely on modeling the behavior (e.g., the power consumption, electro-magnetic emissions or computing time) of a portion of a computing device acting on a small amount of secret data, e.g., one byte, for all the possible guesses of such a secret value. The device behavior is then measured and compared against the guess-dependent models, revealing which one is actually correct, as it will be the only one fitting the measures. Traditionally, side channel attacks are split in two categories, i.e., *profiled* [14], [28] and *non profiled* attacks [12], [37]. The former ones derive the device dependent model in a data-driven fashion, from another instance of the device under attack over which the attacker has full control. The attacker collects a large amount of measurements from the said additional instance of the device at hand for every possible value of the secret to the end of building templates of the device behavior. These templates are subsequently employed to match the measurement taken on the attacked device. By contrast, in non profiled attacks, the attacker devises a synthetic model

of the behavior of the device from a closed-form model instead of regressing it from the data.

In this work, we focus on counteracting non profiled side channel attacks that exploit either the power consumption of the computing device or its near-field electro-magnetic (EM) emissions as the informative channel to be measured. We note that the two side channels at hand provide closely related information, unless analog power decoupling approaches [5], [19] or EM signature reduction techniques [18] are used.

Countermeasures against non profiled side channel attacks aim to reduce the signal-to-noise ratio of the side channel to the point where the number of measurements an attacker must make to derive the secret information goes beyond her capability or the time required to perform the analysis goes beyond the expected lifetime of either the device or the secret information itself. This approach has a more limited effect in case the attacker is employing profiled attacks, as she will implicitly include the device noise within her template and, provided that an arbitrarily long profiling activity is feasible (as she controls the profiled device), she will succeed. The topic of counteracting profiled side channel attacks was recently tackled in [8], where a countermeasure which can be paired to the ones for non profiled attacks is provided.

Countermeasures against power consumption based or EM emissions based side channel attack are usually categorized into *hiding* techniques, *masking* techniques and *(code) morphing* techniques. Designers tend to apply one or more countermeasures to the considered device, to exploit the consequent synergy in signal-to-noise ratio reduction.

Hiding techniques aim at decreasing the informative content of each measurement sample via one or more of the following: *i)* changing the physical design of the computing device to reduce the emitted information [18], [38], *ii)* adding noise to the side channel via an additional randomized power consumption source [19], *iii)* randomizing the order in which data independent operations are executed, in turn forcing the attacker to cope with the lack of knowledge on when the computation being modeled in the attack takes place [20].

Masking techniques exploit a randomized redundant encoding of the data [34] that is processed in order to break the link between the actual power consumption of the computing device and the intermediate values of the computation [44]. Such an encoding is devised so that it is possible to perform the entire computation on encoded data without decoding it first; hence, the randomized encoding is removed only on the computation results. The only way for an attacker to circumvent the masking countermeasures is to exploit simultaneously the information coming from multiple operations acting on different portions of the randomly encoded data, and devise a way to recombine such information into a value which does not depend on the added randomness.

Code morphing techniques [2] hinders the ability of the attacker to model the side channel behavior of the device through randomizing *how* the sensitive computation itself is performed. The technique (awarded as one of the Top Picks in Hardware and Embedded Security in the period 2012-2017 [3]) is highly effective, as it provides a continuously moving target for the attacker trying to model the side channel behavior of the device. This randomization is achieved either by means of periodic dynamic recompilation of the code being executed [2], [11], [15], [16], or by emitting multiple (semantically) equivalent code segments and randomly picking the executed one [4]. An approach relying on dynamic code recompilation allows a larger amount of flexibility in the differentiation of the generated code portions, e.g., allowing the randomized rescheduling of the operations, at the cost of requiring a writeable code segment (which may not be available, e.g., on microcontrollers).

A. CONTRIBUTION

We propose the first, to the best of our knowledge, architectural design providing transparent code morphing support to thwart power consumption and EM emissions side channel attacks. We integrate in our design also the memory operation protection strategy proposed in [4]. We implement our integrated code morphing engine in the OpenTitan System on Chip (SoC) [42], as it is a typical CPU design to be employed in secure root-of-trust modules, which are usually constituted of simple, in-order CPUs [10]. Considering the resource overheads with respect to its simple 2-stages in-order design, the OpenTitan CPU provides a reasonable worst-case scenario for the relative area overhead to be expected when adding our solution to any other existing CPU design. We will refer to our solution as the Metis CPU and Metis SoC, from the Greek name of the shape-shifting titan.

B. RELATED WORK

Providing side channel countermeasures transparently to software developers, via a microarchitectural design has already seen research efforts regarding the use of the hiding and masking techniques. In particular, to the end of hindering the attacker's ability to find out the time instant where a sensitive operation is being computed, the authors of [10] proposed a generic custom hardware unit to randomize the execution order of independent instructions. In [26] the authors propose a redesign of the datapath of the V-scale RISC-V core, to transparently introduce masking countermeasures into the computation. Their approach implements in hardware the redundant randomized operand encoding mandated by the masking technique as well as the corresponding redundant computation logic, while preserving the CPU ISA. Their approach tackles the portions of the RISC-V core comprising the datapath, which were confirmed to be the ones providing the most significant leakage in [6]. In Metis, we implement our code morphing methodology so that the entire set of actions of the datapath on the processed data are subject to code morphing, effectively improving the attack surface coverage with respect to the design proposal of [33], where dedicated randomizers were prefixed to the execution stage. A further approach to the transparent application of masking and shuffling countermeasures at a microarchitecture level is reported in [13], where the authors propose the masking of

the CPU datapath together with the transparent introduction of random delays inserting randomly pipeline stalls. Another approach at transparently providing masking countermeasures at architectural level is the proposal of [35], which devise an ISA extension for accelerating the redundant randomized computations required in masking countermeasures, through a bitsliced representation of the data and dedicated instruction to encode and decode the bitsliced representation itself. A different approach to dynamically change the power consumption of the functional units performing the computation is presented in [51], where the authors propose to employ a reconfigurable functional unit paired to a classic ALU to perform the sensitive operations of cryptographic algorithms. A noteworthy and successful application of the principle of continuously randomizing information related to the computation, albeit to a different security domain, is presented in [23]. The authors propose Morpheus, a CPU modified to continuously alter the value of system level metadata (e.g., code pointers) to the end of presenting a moving target to an attacker exploiting software vulnerabilities arising from the application of control flow redirection techniques (e.g., buffer overflows, return-to-libc attacks, return oriented programming). This application of the randomization principles falls in line of previous successful attempts at preventing timing side channel attacks through code randomization as reported in [17].

II. BACKGROUND

In the following, we provide a background on power and EM-emissions side channel attacks and focus on the code morphing countermeasure. Finally, we introduce the main features of the OpenTitan SoC [42], on which we built Metis.

A. POWER AND EM SIDE CHANNEL ATTACKS

Performing a non profiled side channel attack, relying on either the power consumption of the device or its EM emissions relies on a three phase process: *i)* select an intermediate value of the computation involving a small amount of secret information (e.g., one byte) and model the power consumption of the execution of an operation involving it, for every possible guess of the secret value; *ii)* measure the power consumption of the device when it is performing the modeled operation; *iii)* compare the key dependent model of the behavior with the actual measure. Typical models for the power consumption of the device include the *Hamming weight* of the unknown secret dependent value (which assumes the switching of a sequential element from an all-zero reset state to the value being stored), or the *Hamming distance* of two secret dependent values stored in consecutive clock cycles in the same sequential element.

The said attack process operates under two idealizations that need to be managed in practice. The first assumption is that the measurements are noise-free, and that it is possible to measure the power consumption of the component operating on the secret value alone. Since this is not the practical case, the attacker copes with the intrinsic noise of the measurement taken through measuring the behavior of the computing device for a statistically significant amount of times and employs a statistical tool to determine the match between the synthetic, key dependent models and the measurements themselves. One of the most common statistical tools in this context is the Pearson's linear correlation coefficient [12], which was proven to be the optimal tool [27] under the hypothesis that both the data related power consumption of the device is proportional to the number of switching components in the device itself, and the noise can be modeled as a zero-mean additive Gaussian noise. Correlation Power Analysis (CPA) [12] considers the set of power consumption measurement samples as the realizations of a random variable $\mathcal{Y}$, and the values of the predicted power consumption as the realizations of another random variable $\mathcal{X}_k$, with k being a fixed value of the secret parameter to be found. The two series of realizations are obtained by applying in the same order the same known inputs to the target device and to an algorithm computing the target intermediate value of choice, respectively. Computing, for all the possible values of the secret parameter $k \in K$ the sample estimate of the correlation coefficient $\rho(\mathcal{X}_k, \mathcal{Y})$, and determining the value $\bar{k}$ for which $\rho(\mathcal{X}_{\bar{k}}, \mathcal{Y})$ is maximum allows the attacker to deduce that $\bar{k}$ is the actual value of the secret parameter.

The second assumption is that the attacker knows the exact moment when the modeled operation takes place. Since this is not the case, the attacker samples the power consumption of the device over a time interval which includes with certainty the instruction being modeled, memorizing a time series of the power consumption called *power trace*, or simply *trace*, for each execution. After memorizing a set of power traces, the statistical tool is applied to the set of samples which correspond to the same instant in the power traces. This approach requires that the samples in the same instant of the traces actually correspond to the power consumption of the same operation, i.e., the traces should be perfectly aligned. This requirement is intentionally violated by the hiding countermeasure known as *shuffling*, which consists in a randomized rescheduling of the order of the independent operations being computed at each execution, in turn breaking the aforementioned trace alignment.

B. CODE MORPHING

Realizing a code morphing countermeasure through dynamic code recompilation [2] substitutes each instruction of the original implementation of a program with a randomly chosen instruction sequence which has the same semantics, i.e., starting from the same input values, computes the same output value(s) as the original instruction. To this end, the main practical observation is that memorizing the equivalent instruction sequences for all the possible instructions in the ISA, also considering all the possible input/output register combinations, requires an excessive amount of resources. As a consequence, the code morphing approach, depicted in Figure 1, is realized by a compile-time phase and a runtime phase. The first phase, known as the *tile generation process*,

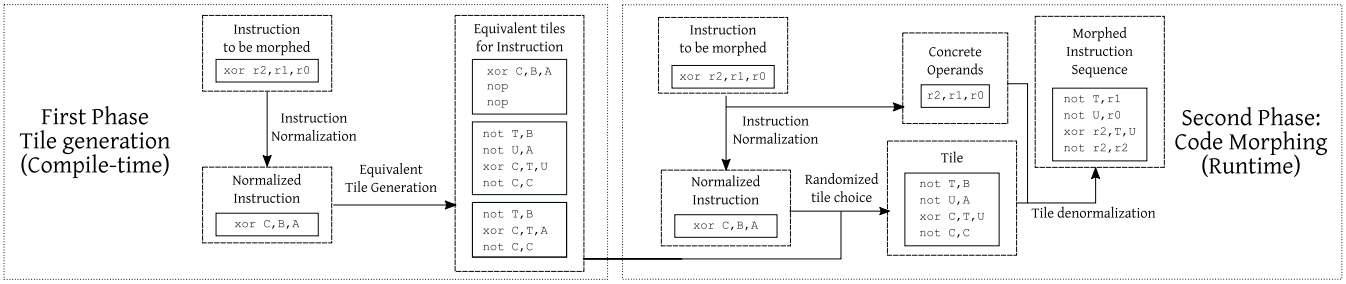


FIGURE 1. Graphical representation of the two phases of the code morphing approach. The first offline phase (left) generates a set of equivalent tiles for each instruction, which are employed in the online phase (right), where the code morphing takes place.

analyzes the instruction sequence to be morphed, and identifies all the instruction opcodes present in the said sequence. Subsequently, for each instruction opcode, a normalized representation of the instructions containing it is computed. The normalization process removes from an instruction both the concrete ISA register names and immediate operand values replacing them with symbolic labels.

The process yields a set of normalized instructions smaller than the set of original instructions. Indeed, multiple instructions sharing the same opcode and having the same number of distinct registers in the same operand order are replaced by the same normalized instruction.

For each normalized instruction, multiple equivalent normalized instruction sequences, called *tiles*, are created by the designer instantiating the morphing countermeasure. Each tile performs the same computation of the former normalized instruction. Only a subset of the ISA instructions may require morphing, as they are involved in secret information processing: the choice of which instruction should be morphed is delegated to the side channel security expert [2].

In particular, a tile may realize the computation of the former normalized instruction by applying a masking strategy or by shuffling the initial instruction with *dummy* ones. Tiles are stored in a lookup table indexed by the former normalized instruction.

The second phase, i.e., the actual code morphing at runtime, employs the compile-time generated normalized tiles to dynamically replace the instructions in the code. To this end, in [2] a small just-in-time compiler is realized and invoked before each execution of the function to be protected. The just-in-time compiler iterates over all the instructions to be morphed and normalizes them following the same labelling criterion employed in the offline phase, memorizing the actual operands of the instruction to be morphed. Following this, the just-in-time compiler randomly chooses from the normalized tile table one of the tiles corresponding to the instruction to morph, and de-normalizes it, i.e., performs a local register allocation, taking care of the eventual clobbering and de-clobbering required by additional registers. The product of the said de-normalization process is a fully functional instruction sequence which has the same computational effect of the original instruction to be morphed.

The only instructions that cannot be replaced with a semantically equivalent tile are the ones corresponding to `load` and `store` operations, bar for the addition of dummy instructions and the shuffling of the resulting sequence. To this end, in [2] memory operations are protected via shuffling. In particular, memory accesses amenable to a randomized rescheduling are located in the compile-time phase of code morphing, and their run-time execution order is randomized with a vector of indices which is permuted at run-time. An alternative approach is the one adopted in [4], where the authors protect `load` and `store` operations through masking of the loaded and stored values. The authors exploit the presence of a code morphing countermeasure to strengthen the load/store masking protection against the attack proposed in [54].

C. THE OpenTitan SYSTEM ON CHIP

The OpenTitan SoC is designed to be an open source platform for a silicon root-of-trust for a variety of systems (e.g., acting as a smartcard or as a single-sign-on physical token), therefore well representing the typical target of side channel attacks [47]. OpenTitan is endowed with a reliable, production grade development environment, and the SoC includes basic peripherals such as UART and SPI interfaces connected via a TileLink-Uncached Lite crossbar network, and a complete JTAG network.

The OpenTitan SoC is based on the Ibex CPU, an in-order 32-bit RISC-V processor, designed to be deployed either on FPGA or as an ASIC implementation. Its design follows a Harvard architecture, with a two stage pipeline, depicted in Figure 2, composed by an Instruction Fetch (IF), an Instruction Decode and Execute (ID-&-EX) stage and neither instruction nor data caches [40]. Ibex is compliant with the RISC-V ISAs I (base integer operations), M (standard integer multiplication and division) and C (16-bits compressed instruction encoding). Optionally, it can be configured to replace the RISC-V ISA I (a.k.a. RV32I) with the E one (a.k.a. RV32E), implementing a reduced Register File (RF) with only 16 registers, for resource constrained designs.

1) INSTRUCTION FETCH STAGE

Instructions are retrieved from a three instruction deep prefetch FIFO buffer filled with 32-bit of data on each clock

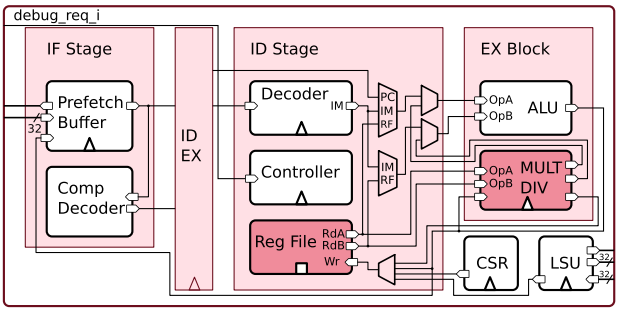


FIGURE 2. The two-stage pipeline of the Ibex CPU powering the OpenTitan SoC. The components in dark red can be configured at design time to support the M and E RISC-V ISA extensions.

cycle, bar memory throughput bottlenecks. If the buffer is empty, a pass-through logic forwards the instruction directly to the pipeline registers. The IF stage of the pipeline decompresses 16-bit instructions into their 32-bit equivalent ones to relieve the complexity from the decode unit. On the issue of a new instruction, a single cycle pulse signal is generated, allowing the sequential logic circuits of the ID-&-EX stage to register and process the new data once.

2) INSTRUCTION DECODE AND EXECUTE STAGE

This stage contains a large part of the logic of the processor, which parses the issued instruction and actuates the required operations. A dedicated module checks the permissions of memory accesses, while a controller is coordinating the entire stage. *Controller.* The controller is managing the Program Counter (PC), dealing with all the situations that need particular care. On branch and jump operations, it sets the destination address from the output of ALU. Upon interrupts or exceptions, according to the actual configuration, manages the change of context by setting the correct Control Status Register (CSR) values as well as the startup and flush of the pipeline, the handling of sleep, wake-up and debug requests, and the setting and release of the instruction fetch freeze. *Register File.* The RF is designed as a dual-port RAM, with a single write port, and can be implemented by flip-flops or latches; the former are used by simulators and the latter by ASICs. FPGAs can use flip-flops and/or device-specific distributed memory (LUTRAM). Provides access to 31 or 15 32-bit width registers, depending on the used ISA. Register `x0` is hardwired to the zero value. *Decoder.* The fully combinatorial decoding logic is in charge of preparing all the muxes connecting the RF ports, the operands fed into the Arithmetic Logical Unit (ALU) and the destination of the write back actions. A small sequential logic circuit is used to keep the state of multi-cycle operations, such as memory accesses, multiplications, divisions and special control instructions.

a: EXECUTE BLOCK

The ALU is designed to compute all RV32I instructions, whilst an optional Multiplier/Divider block is provided to comply with the RV32M instruction set. The adder circuit is employed also to compute target addresses of `load`/`store` and `branch` operations.

b: LOAD/STORE UNIT AND PHYSICAL MEMORY PROTECTION UNIT

The Ibex Load/Store Unit (LSU) manages the data bus connecting the external memory, requesting a read or write operation at a word-aligned address. Differently from what specified in the RISC-V documentation, it can also manage a mis-aligned request by splitting it into two consecutive aligned ones. Data coming from the external data bus is directly routed to the RF write port. thus, the delay contribution due to pipelines in the Ibex core is kept at its minimum. In FPGA implementations, the minimum overall cost of a memory operation is 5 clock cycles.

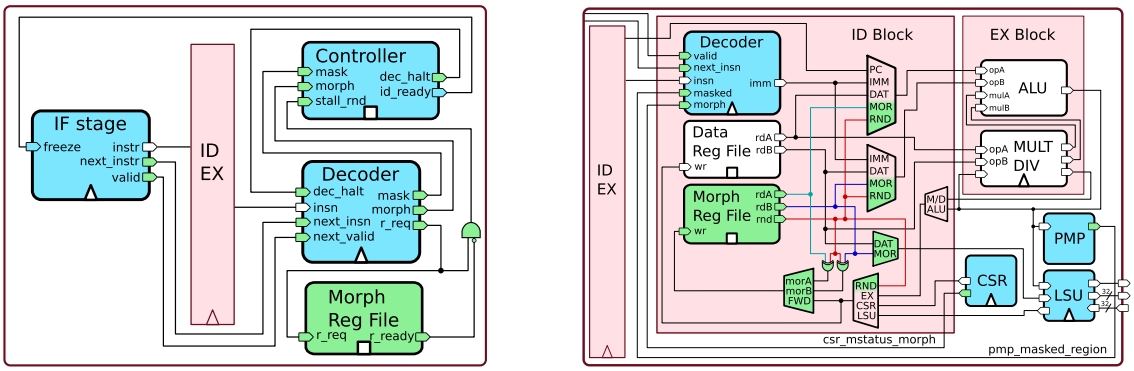
The Physical Memory Protection (PMP) unit enforces read, write and execute permissions on a number of memory regions up to 16. After a memory address is computed by the ALU, it is checked against the defined rules and an access error is asserted if an access violation will take place. This module is instantiated for both the fetch and load/store units, and the check result is used to gate the external data request signal. It is possible to define up to 16 memory regions making use of the configuration registers pmpcfg0-pmpcfg3 and pmpaddr0-pmpaddr15, specifiying either the beginning and end of a region employing two registers, or, employing a single register either a single 4-bytes region or a power-of-2 sized region, both 4-bytes aligned.

III. A CODE MORPHING CPU

In this section, we describe our Metis micro-architectural design to provide transparent code morphing to a designer-chosen subset of the Ibex ISA, together with the randomized mask refresh technique, proposed in [4], for accessing lookup tables. We describe the modifications to the IF and ID-&-EX stages, together with the required additions to transparently preserve the CPU state while morphing instructions. Subsequently, we describe our modification to the LSU and memory interface to support the randomized mask refreshing. The details of the required Random Bit Generator (RBG), which can be implemented as either a Cryptographically Safe Pseudorandom Bit Generator (CS-PRBG) or a True Random Bit Generator (TRBG) are outside the scope of this work. Indeed, there is a significant corpus of literature tailoring the optimal solution to a desired FPGA or ASIC target.

A. THE METIS INTEGRATED MORPHING ENGINE

We realize code morphing at ISA level designing the Metis CPU so that the second phase of the morphing approach described in Section II is performed by the ID-&-EX stage of the pipeline. Figure 3 shows the proposed design highlighting in cyan the portions which have been modified, and in green the modules which have been added.



(a) Full Metis pipeline

(b) Metis decode and execute stage.

FIGURE 3. Stages of the Metis pipeline with differences from the original Ibex pipeline higlighted. Green components are newly added, blue components have been modified. Red buses are carrying random data being read from the integrated RNG.

1) CONTROLLER

The Metis pipeline, depicted in Fig. 3a is designed to manage the pipeline stalls required for transparent code morphing. The Controller receives three signals, the `mask` and `morph` signals from the Decoder module, and the `stall_rnd` one from the RNG. The two former signals determine when the IF stage should be frozen due to the processing in the ID-&-EX stage requiring more than a single clock cycle to manage either the tile of an instruction to be morphed or a masked load/store operation. The third signal allows the Controller to stall the pipeline in case a fresh value from the RNG is needed and not yet available. To manage debug requests, the Controller signals to the Decoder, via the `dec_halt` line, that a debug request was made. The Decoder module completes the morphing of the current instruction, reverts to the regular execution mode and leaves Metis ready to execute the first debug instruction. Since the `dec_halt` line is kept high during the entire debug session, the Controller inhibits the start of further morphing or masking actions, thus maintaining compatibility with external debuggers.

2) INSTRUCTION FETCH

The IF stage in Fig. 3a differs from the one of the original Ibex CPU shown in Fig. 2 in the addition of combinatorial path (named `next_instr` in Fig. 3a), which allows the Decoder in the ID-&-EX stage to establish, at least one cycle in advance, if the instruction currently in the IF stage is going to be morphed or not. The signals on the said combinatorial path are gated by a further `valid` signal, also starting from the IF stage, to avoid the propagation of spurious data into the Decoder when either the subsequent instructions in the prefetch buffer are not going to be executed due to a modification of the control flow or a debug interrupt request has to be managed.

3) CONTROL STATUS REGISTER

To allow the code morphing actions to be selectively enabled, the mstatus register in the CSR module (see Fig. 3b) is modified as shown in Fig. 4. Specifically, the mstatus register

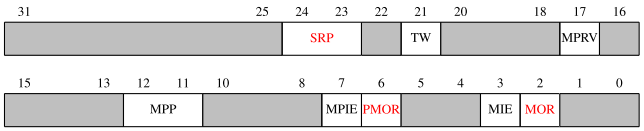


FIGURE 4. modified mstatus register, containing the *morphing execution* bit at position 2, and its previous value at position 6, and the *Shares Refresh Probability* at positions 23 and 24.

includes a MOR bit indicating when the Metis CPU is in code MORphing mode. In particular, the MOR bit is reset to zero on reboot and toggled at run-time via a `set` instruction. In case an exception or an interrupt request takes place when Metis is operating in morphing mode, the MOR value is stored into the PMOR bit of the same register and the morphing logic is disabled. The return to the original status, through the execution of an `mret` instruction, automatically moves back the PMOR value into the MOR bit, restoring the correct execution mode. In case the software toggling of the morphing actions is not desired, as in the case of extremely demanding security environments (e.g.,smart cards) the control bit can be simply tied to the `enabled` value at design time.

4) MORPHING REGISTER FILE

The main challenge to support the execution of tiles employing a larger number of registers w.r.t. the instructions they morph is to cope with possible register clobbering. To this end, we include in Metis a Morphing Register File (MRF), which provides temporary registers to de-normalize the tile at hand. The MRF is designed with three-read ports and one write port to include a maximum number of 16 registers, which are enough even for the most demanding instruction tiles. The first register of the MRF, `MR0` is hardwired to the zero value. Two of the read ports are connected to the general purpose registers in the MRF, while the third is connected to a RNG. In our experimental evaluation of Metis, we consider an RNG providing a 32-bit word of random data every 2 clock cycles, as it is customary in commercial

4	3	2	1	0	
0	MRF reg. address				} morphing register file addressing
1	0	0	0	1	} SRC1 field of original instruction
1	0	0	1	0	} SRC2 field of original instruction
1	0	0	1	1	} DEST field of original instruction
1	1	0	0	0	} random value from MRF RNG

FIGURE 5. Encoding of the information in the register descriptor fields of instructions in tiles. Symbolic labels for original operands, or a fresh random, have encodings with a leading 1, while the ones of actual MRF registers have a leading 0.

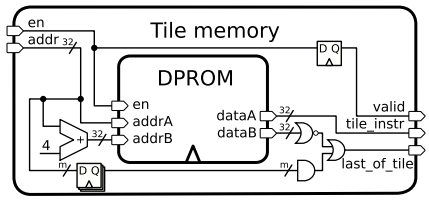


FIGURE 6. *Tile* memory module within the Decoder module.

microcontrollers [53]; however, the Metis Controller is designed to stall the execution for longer or shorter time periods, depending on when the randomness is available for readout. The MRF read and write ports are connected to the input and output operand muxes of the functional units in both the EX block and the LSU, providing the same access latency as the main RF. The reason for dedicating a specific read port to the RNG lies on the choice to read two operands, plus the required fresh randomness, during the processing of a load/store masking operation without performance penalties.

5) TILE MEMORY

The Decoder module contains a dedicated storage for the instruction tiles, called *tile memory*, represented in Figure 6, which is directly accessed by it to minimize latencies. Our design does not constrain the tile memory as a read-only memory; indeed, in a practical design, it can either be in the form of a ROM or of an SRAM if a dynamic method to update the tile set is desired. In FPGA targets, the tile memory is implemented employing lookup tables (LUT), to provide effective readout of a normalized instruction from a tile in a single cycle. Each row of the tile memory contains a single normalized instruction; the overall organization of the tile memory exhibits n groups of rows, each of which corresponds to a different tile that in turn is composed by at most m RISC-V instructions. The designer may pick any choice of n and m, independently, as powers of two: the rationale of this restriction is to simplify the addressing logic. If a tile contains less than m normalized instructions, the remaining tile memory rows are set to zero. This in turn allows the tile memory module to detect whenever the last normalized instruction of a tile is readout and concurrently assert a `last_of_tile` output signal. We represent a single normalized instruction in the canonical RISC-V instruction encoding in the tile memory by replacing the main RF addresses with the binary encoding of our symbolic labels as depicted in Figure 5. In particular, all the encodings starting with a null bit employ the remaining 4 bits to indicate a register file address in the MRF. This allows a complete normalization of the *R-type* (register operand type) instructions of the RISC-V ISA. The normalization of an immediate value in *I-type* (immediate operand type) instructions is done as follows: a null value in the immediate field of a normalized instruction indicates that it equals the original immediate value of the un-normalized instruction. By contrast, any non-zero value in the immediate field of a normalized instruction is employed to denote an additional immediate value employed in the tile at hand, which has no relations with the immediate value (if any) in the original (un-normalized) instruction. This choice allows full expressivity in the tiles is possible to employ the `MR0` register in the MRF, hardwired to the constant value 0, if a zero immediate value needs materialization during a tile computation. The binary encodings of a normalized instruction that starts with a set bit are mapped to symbolic labels for the first and second inputs of the instruction to be morphed, to the output of the instruction itself and to a fresh random value to be read from the MRF RNG. To reduce the designer effort in creating new tiles, the specification of the tiles themselves is provided in an HJSON structured format, from which the translation in binary format is automated by means of the `gcc` compiler equipped with the RISC-V backend, outputting a Verilog VMEM formatted document to be included in the design with a `$readmemh` Verilog statement.

6) DECODER MODULE

The Decoder module of Metis is responsible for the computations related to the run-time code morphing process and for driving the PMP mechanisms. To this end, a sequential logic component of the Decoder module tracks whether it is operating in *normal execution*, *morphing execution*, or *memory masking* mode. We note that the said modes of operation are mutually exclusive. The mode of operation determines whether the original Ibex combinatorial decoding logic should process the output of the IF/ID interstage register, the morphing tiles or a memory masking operation. Since the Metis Decoder module is able to act as alternate source of instruction to be issued, the *instruction_new* signal was modified so that a pulse can be generated on it also by the morphing and memory masking logic. To illustrate the behavior of the module in morphing mode, we consider the example shown in Fig. 7, where a sequence of instructions `A`, `B`, `C`, `D` are processed assuming that `B` and `C` require morphing, while `A` and `D` do not.

The `next_instr` lines originating from the IF stage provide the value of instruction `B` to the Decoder module the cycle before the first de-normalized instruction of the randomly chosen tile corresponding to `B` is issued (cycle -1 in Fig. 7). The Decoder module matches the opcode (and the `funct` field(s)) of instruction `B` detecting if it is going to be morphed, and signaling to the Controller to freeze the IF from the next cycle. If the `next_instr` lines provides an instruction that requires morphing, the Decoder module also randomly selects the normalized tile to be executed

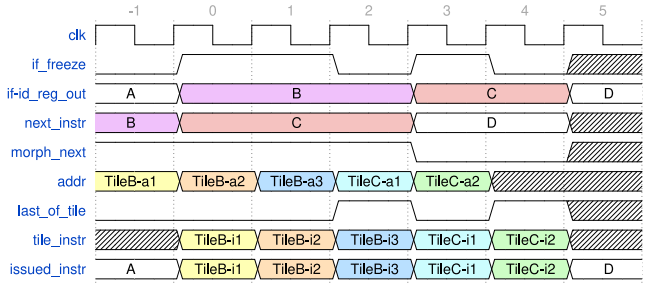
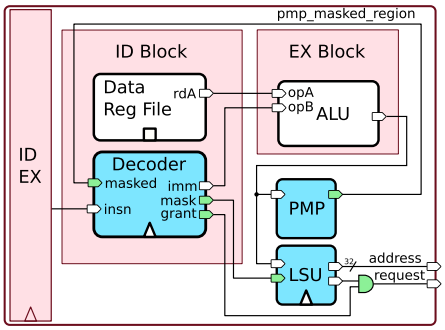


FIGURE 7. Timing diagram of Metis morphing instructions `B` and `C`, executed after `A`, and before `D` which do not need morphing, with two tiles composed respectively by 3 and 2 instructions.

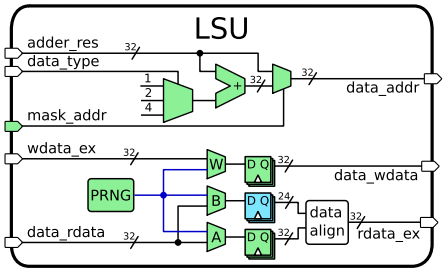
and computes the address of the first normalized instruction in the tile memory. The random selection of the tile is performed as follows: the Decoder module employs an associative table storing the base address `base_addr_I` of the tile memory location where the sequence of n possible alternative tiles resides, for any given normalized instruction `I`. We recall that each one of such alternatives is stored in m rows of the tile memory. A tile is randomly selected extracting a $\log_2(n)$-bit value r from a dedicated small RNG, and composing the address of the first instruction in the tile to be de-normalized and run as $(\texttt{base\_addr\_I} + r) \times m$. In Fig. 7, the first address of the first instruction of the randomly chosen tile corresponding to instruction `B`, `TileB-a1` is made ready on the `addr` signal at cycle -1. We assume that the randomly selected tile for `B` is three instructions long. In the following clock cycle, the decoding unit de-normalizes on the fly and issues the instruction `TileB-i1` at the tile memory address `TileB-a1`. The register de-normalization is performed through combinatorial blocks which employ the symbolic labels in the normalized instruction in the tile memory to connect the appropriate functional unit in the EX block to either the MRF, the original RF or the RNG. The decoding unit de-normalizes and issues all the normalized instructions in the tile, until the `last_of_tile` signal is raised. The rise of the `last_of_tile` signal also allows the Controller module to unfreeze the IF stage, thus exposing the value of the next instruction, `C`, on the output of the IF/ID inter-stage register at the next cycle (cycle 3). Since `C` also needs morphing, the Decoder module performs, at cycle 2, the same procedure which was performed at cycle -1, i.e. randomly selects a tile for `C` and computes the address of the first normalized instruction of the tile `TileC-a1`. The de-normalization and issuing of the two instruction constituting the randomly chosen tile for `C` is done in cycle 3 and cycle 4. At cycle 4, the Decoder module reads the instruction `D` from the `next_instr` lines originating from the IF, detects that no morphing is needed, and resumes the normal operation mode, issuing instruction `D` unmodified at cycle 5.

B. PROTECTING MEMORY ACCESSES WITH MASK REFRESHING

Transparently providing protection to the memory operations is realized in Metis through automatically loading and storing masked values for a set of sensitive memory locations. To make the Metis CPU aware of which portions of the memory are in need of masked loads and stores, we exploit the existing PMP unit, extending its memory attributes (read, write and execute) with a further `masked` attribute employing the bit 5 of the pmp*x*cfg register, which is reserved for extensions. By default, the masked attribute is not set. Similarly to [4], we rely on the software developer to tag in the high level source code the sensitive memory locations and on the compiler to emit the instructions to toggle the memory protection attribute on the memory portions which require masking. A signal line is connected from the PMP unit to the Decoder module, to notify the latter of an attempt to access a memory location which must be protected via masking. Since the entire memory protection mechanism is contained in the Metis PMP unit and LSU, the remaining part of the pipeline is kept frozen while a protected load/store operation is performed. The PMP unit prevents a memory access from taking place as the request going from the LSU to the main memory is gated with a grant signal provided by the PMP (Figure 8a). The design of Metis supports a so-called *first order* masking, i.e., each value v to be protected is replaced by a pair of values, s_1, s_2, known as *shares*, $\langle s_1, s_2 \rangle = \langle v \oplus r, r \rangle$, where r is a random value. The design can be easily extended to *higher order* masking techniques, which employ the same principle, increasing the number of shares and the required random values. Providing mask refreshing in this context indicates that, each time a masked value is accessed, a fresh random value should be added to both of its shares. However, employing a deterministic mask refreshing scheme was exploited in [54], which prompted for a random mask refreshing scheme to be integrated in the code morphing approach described in [4]. We integrate in Metis such a randomized mask refreshing approach. Integrating the support for first order masking in the memory subsystem of Metis requires to manage two needs: *i)* automate the loading and storing of masked values; *ii)* avoid that the switching activity of the memory bus or main memory depends on an unmasked value. The first need is solved through a modification of the Ibex LSU and Decoder module. To automate the loading and storing of masked values, Metis assumes that the memory layout of their shares is such that the shares are stored contiguously in memory. This approach allows to automate the readout (resp. storage) of the shares of a value via two consecutive load (resp. store) operations. To this end, the LSU in Metis, (Figure 8b) is endowed with a dedicated adder to compute the address of the second share of a masked value, given the address of the first share (`addres_res`) and the data type. Executing a `load` instruction on a masked value requires the recombination of the two shares, i.e., the computation of their bitwise `xor` before the loaded value can be employed in the computation. To this end, the LSU stores the loaded shares in two MRF dedicated registers; the loaded shares are `xor`-combined by the ALU and stored in the destination register of the `load` instruction in the main RF. Symmetrically, the store operation loads the second share,



(a) Memory access protection mechanism structure (new components in green, modified ones in cyan)



(b) Metis LSU: signals with *data_* prefix belong to the data bus interface, others are directed to the ID-&-EX

FIGURE 8. Metis modules implementing masked load/store operations and randomized mask refreshing.

containing a random value alone in the MRF, `xor`-combines it with the value to be stored, and stores the new first share at the destination address of the store.

For each load-store operation, Metis performs a mask refreshing depending on the outcome of the extraction of an l-bit random number coming from a dedicated RNG. In our design, we employ a $l = 2$ bits random number, allowing the designer to select a refresh probability among $\{0, 0.25, 0.5, 0.75\}$. To reduce the computational overhead of masking, we perform the mask refreshing of the first share during load operations in parallel with the second load operation, exploiting the availability of the first share to hide the latency of the required `xor`. Willing to avoid that the switching activity of the bus or of the main memory leak information on the unmasked value when the shares are being loaded, it is important to observe that loading in a sequence the two shares $\langle s_1, s_2 \rangle = \langle v \oplus r, r \rangle$ of the masked value leads to a switching activity proportional to the bitwise `xor` of the two shares, that is $r \oplus (v \oplus r) = v$. Such a switching activity would in turn allow an adversary to perform a side channel attack, as the power consumption during the second load operation is proportional to the unprotected value v. In Metis we prevent such a fact from happening, employing an approach which does not depend on the implementation of the main memory, nor on it having a single buffer for read and write operations or two separate buffers. In our approach, we insert, between the two loads of two shares of the same protected value, a store operation and a load operation, both operating on the address of the first share, employing the constant word 0. This, in turn, results in a switching activity proportional to $(v \oplus r)$ during the further store operation, and to r during the loading of the second share, in turn preventing an attack on the unprotected value v from succeeding. To ensure that the switching activity of the memory bus matches our expectations we added two registers on the (write-to-memory) `data_wdata` and (read-from-memory) `rdata_ex` ports.

IV. EXPERIMENTAL EVALUATION

In this section we report the results of experimental evaluation of the performance and security of our integrated morphing engine in Metis. We developed Metis starting from the OpenTitan SoC revision available at [43], commit `b6b1c783`, and the revision available at [41], commit `7e22830`, of the Ibex Core. We chose as the target platform the Digilent Arty A7 board equipped with a Xilinx Artix-7 XC7A100TCSG324-1. We chose this target device thanks to the OpenTitan project supporting natively Artix-7 FPGA targets, with a board support package for the Artix-7 based Diligent Nexys Video board. To adapt the OpenTitan design to the Arty A7 board, we reduced the RAM in the SoC down to 128 kiB and modified the XDC pin constraint file to fit the pinout of the Artix-7 XC7A100T. The reduction in available RAM did not have any impact on our benchmark campaign, indeed leaving enough RAM to run the entire CoreMark benchmark binary [22]. From now on, we refer to this minimally adapted SoC design as the *unmodified* SoC. The synthesis, mapping, placing and routing was accomplished with the Xilinx Vivado Suite ver. 2018.3, targeting a 48 MHz clock frequency for the unmodified SoC.

We considered two variants of the Metis SoC, endowed with different morphing tiles modifying the computation of the `xor`, `and` and `slli` instructions. These instructions were experimentally found to be the ones with the most significant side channel leakage, following the approach in [2], [4], in addition to the `load-store` instructions, which are protected by default in our approach. We benchmarked our approach on two set of tiles: the Shuffling Tiles (ST) set, and the Morphing Tiles (MT) set. The ST set contains two variants for each of the aforementioned instructions constituted by the original instruction itself and up to 3 dummy operations. The aim of the evaluation on the ST set is to quantify separately the amount of protection provided by the intrinsic hiding component of the code morphing countermeasure. The MT set contains 8 alternative tiles for each of the three instructions, `xor`, `and` and `slli`. Each alternative tile contains from 0 to 4 dummy normalized instructions. The `xor` tiles compute the result of the operation $r \leftarrow a \oplus b$ as either $r \leftarrow (\overline{a} \oplus \overline{b})$, or $r \leftarrow \overline{(\overline{a} \wedge b) \vee (a \wedge \overline{b})}$, or the instruction itself. The `and` tiles compute the result of the operation $r \leftarrow a \wedge b$ as either $r \leftarrow a \wedge b$ or $r \leftarrow \overline{(\overline{a} \vee \overline{b})}$ filling the destination register with fresh randomness. The `slli` tiles interleave the

TABLE 1. FPGA Resource utilization post P&R of Metis compared to the OpenTitan SoC and Ibex CPU. The usage DSPs, IOs, BUFGs, and PLLs is unchanged between the SoCs, while no change in the use of BRAMS occurs between the CPUs.

(a) Resource occupation for the CPU

configuration	LUT as logic	LUTRAM	FF
Ibex	6,356	48	1,862
Metis-ST	7,605	96	2,283
Metis-MT	7,725	96	2,298

(b) Resource occupation for the entire Metis SoC

configuration	LUT as logic	LUTRAM	FF	BRAM
Available	63,400	19,000	126,800	135
OpenTitan	20,952	4,140	8,454	83.5
Metis-ST	25,864	12,380	8,910	67.5
Metis-MT	25,925	12,380	8,925	67.5

(c) Registers and LUT of the Metis SoC modules. The LUT utilization does not include the ones used as distributed memory

Microarchitecture	decoder		LSU		morphing RF		data FIFO		memory controller		Total	
Component	LUTs	FFs	LUTs	FFs	LUTs	FFs	LUTs	FFs	LUTs	FFs	LUTs	FFs
CPU logic	478	66	259	138	101	34	122	10	989	33	1,949	281
PRNG	5	160	32	1,024	16	512	32	1,024	32	1,024	117	3,744
Total	483	226	291	1,162	117	546	154	1,034	1,021	1,057	2,066	4,025

`slli` with dummy operations and fill the destination register with fresh randomness. As RNG, we employed 32-bit LFSRs, one per bit to be generated, employing primitive connection polynomials over $\mathbb{Z}_2[x]$ to ensure a maximum period of 2^{32} for the generators themselves. While such generators are not CSPRNGs, they provide a statistically sound output that is sufficient to evaluate the security of Metis. Designing compact and secure RNGs is out of scope for this work.

A. HARDWARE RESOURCES AND TIMING

We synthesized Metis employing the `Flow_Alternate Routability` and `Congestion_SpreadLogic_ high` strategies targeting a 40 MHz system clock, whereas the OpenTitan can sustain a 48 MHz clock under default Vivado strategies. We chose to employ the said strategies due to a 70% congestion on horizontal routing lines in parts of the design, which caused a slack drop of about 4.5 ns between signaled critical paths in synthesis and implementation. The most likely cause of the congestion is the relatively large area of the PRNG we chose, which can be reduced in production environments by more aggressively optimized RNG designs [48].

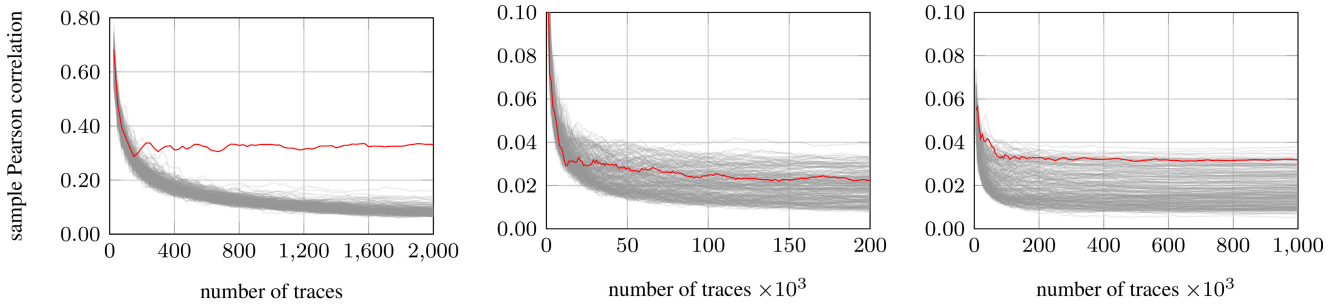
To obtain a fair resource analysis, we also implemented all the compared SoCs using a flattened hierarchy, a 40 MHz clock and the same congestion strategies, allowing us to provide meaningful data in a per-module granularity. We note that the only change in the post P&R results when moving from the original target frequency of 48 MHz to the one of 40 MHz is a single extra LUT employed as logic.

Table 1a shows that the Metis CPU increments the use of FPGA resources w.r.t. the Ibex CPU by 1, 212 LUT (+19%) and 436 FF (+23%). The increase in the use of LUT for implementing RAM (LUTRAM) in Tab. 1a is due to the introduction of the MRF. The tile memory is occupying respectively 37 and 164 LUT for the ST and MT variants. Framing the resource required by the Metis CPU w.r.t. the overall resources required by the OpenTitan SoC, we have a 6% increment for LUTs and 5% for FFs, as a result of the significantly compact design of both Metis and Ibex.

Table 1b allows to compare the overall SoC resource consumption, showing a 24% increase in resource usage for logic resources and a 6% increase for registers. We note that the most significant part of the increase of the LUTs (+8, 192 LUTRAMs, +990 LUTs) is determined by the decision of the Vivado Suite to synthesize the main memory to LUTRAMs instead of the original 16 BRAMs. This decision taken by Vivado is also a likely reason for the increased congestion. As a consequence of the congestion, the comparison of the entire SoC targeting this Artix-7 FPGA is somehow a less meaningful portrait of the actual resource usage. Indeed, we have observed a significant increment of logic resource utilization, ranging from +20% to +45%, in some unmodified modules common to both the OpenTitan SoC and the Metis variants. This inconsistency is likely to be the product of the optimization choices actuated during the place-and-route phases.

Although congestion levels are preventing an unbiased timing analysis for the CPU design alone, we highlight a common critical path that, starting from the registers holding the tile instruction to be decoded, is produced by memory-related instructions. The path includes the computation of the memory address by the ALU, which passes through the introduced adder in LSU, to be then checked by the PMP module eventually triggering an access exception by means of the controller logic. This described path is responsible for a 24 ns delay, explaining the CPU frequency limitation. We note this path that can be considered a false one as no memory instructions are contained in the tile memory in the first place.

Data reported in Tab. 1a and Tab. 1b omit the RNG's contribution, which is instead detailed in Tab. 1c on a per-module basis, for the sake of completeness. To put the resource use of Metis in perspective, the highly area efficient EM suppression technique in [18], targeting ASIC devices, reports a conservative +22% area increment. The work in [26], which provides architectural support for masking, reports an increment of 59% for logic and 85% for registers, when it is protecting against first-order attacks, and the said figures grow almost



(a) Ibex (Unprotected) (b) Metis-ST (Protected) (c) Metis-MT (Protected)

FIGURE 9. Results of performing a CPA against AES-128, targeting the computation of the SubBytes primitive on both the unprotected Ibex and on Metis. The sample correlation of the correct key hypothesis depicted in red, while the sample correlation of all the other key guesses is reported in grey. We note how the correlation of the correct key hypothesis does not rise above the incorrect key guesses in Metis, indicating that the CPA does not succeed.

linearly up to 255% and 358%, respectively, when it is defending against fourth-order attacks.

B. SIDE CHANNEL SECURITY EVALUATION

To evaluate improvement of the security margin provided by Metis, we chose to employ the Measurements-To-Disclose (MTD) metric, i.e., the number of power traces required to be able to extract a secret value via side channel attack, as in [2], [4], [11]. We employed as target cipher the AES cipher [29], with a 128-bit key, and performed a correlation power analysis targeting the computation of the SubBytes primitive of a standard compliant S-Box based implementation. We experimentally validated that the best fitting power consumption model, i.e., the one allowing to retrieve the secret key with the least MTD on the unprotected SoC was the Hamming weight of an output byte of the SubBytes primitive. We report that such a model matches both the computation of the SubBytes primitive, and the ShiftRows primitive, as the latter performs byte-wise permutations of the output of the former.

To measure the side channel leakage of our solution, we employed a Picoscope 5203 digital sampling oscilloscope, sampling at 500 Msamples/s the output of an Agilent INA-10386 wideband amplifier (26 dB gain up to 1.5 GHz) connected to a TekBox EMC near field probe H10 (TBPS01 kit). The probe was placed on the top of the Artix-7 package, locating manually the place were the most significant emissions were present. We set the probability of performing a mask refreshing to 0.25. Figure 9 reports the results of conducting the correlation power analysis against the unprotected OpenTitan SoC (Fig. 9a) and two version of Metis employing respectively the ST tiles (Fig. 9b) and MT tiles (Fig. 9c). The correlation power analysis succeeds in obtaining the correct AES key value starting from 800 measurements, as the confidence intervals for $\alpha = 0.1$ of the correlation of the correct key guess (red) and the highest-correlating wrong guess (black) are disjoint for the first time at the said number of measurements. We note that the fact of the OpenTitan SoC being attackable does not represent a design issue of the SoC itself, as it is not designed for side channel attack resistance of its main CPU. Metis protected solutions successfully prevent the recovery of the correct key value when 200k and 1M traces are employed for the ST and MT tile-sets, respectively. This represents an $>250\times$ and a $>1,250\times$ improvement in the MTD metric w.r.t. a correlation power analysis targeting an unprotected design. We also note that the correlation of incorrect key values is higher than the one of the correct key guess, as a result of the code morphing distortion of the power consumption leakage, coherently with [4].

Comparing security margins, the software code morphing solution in [11] targets a protection exhibiting an MTD exceeding 1M traces and shows a morphing strategy that is effective up to 5M measurements, starting from an unprotected design having leakage at 290 measurements. The software code morphing solution in [4] evaluated on the MTD metric reports a MTD increase of $1,000\times$ for the EM emission analysis. While it is not possible to directly compare different side channel countermeasure techniques, we report the resistance figures of other approaches to integrate them at architectural level to provide a more inclusive overview. The solution integrating masking countermeasures proposed in [26] obtains a boost in MTD metric of at least $50\times$ (from 2M to no leakage appearing at 100M traces). The solution integrating hiding countermeasures in [10], instead achieves a $366\times$ increase in the number of required measurements derived from the decrease in the Pearson correlation coefficient. Finally, the EM emission reduction technique proposed in [18] provides a $> 166\times$ improvement on the MTD metric, showing that no leakage takes place at 1M measurements.

Willing to provide a deeper analysis of the security of the solution proposed by Metis, we report also the results of computing two other side channel security metrics, namely the *success rate* and the *guessing entropy* [52]. The success rate is computed performing multiple side channel attacks on independently collected measurements, with an attack methodology which outputs a ranking of the key hypothesis, from the one most likely to be correct to the least one according to the extracted information. For any given value of the order o between 1 and the number of key hypotheses made in the side channel attack, the success-rate of order o is defined

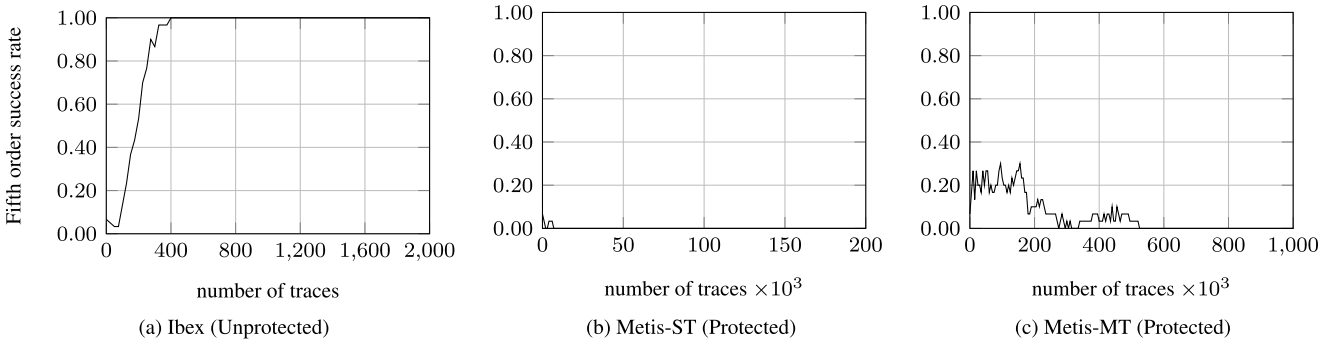


FIGURE 10. Results of computing the success rate of the CPA attacks against AES-128, targeting the computation of the SubBytes primitive on both the unprotected Ibex and on Metis. All the success rates were obtained averaging the success rate on 30 independent experiments, each one performed with a different secret key.

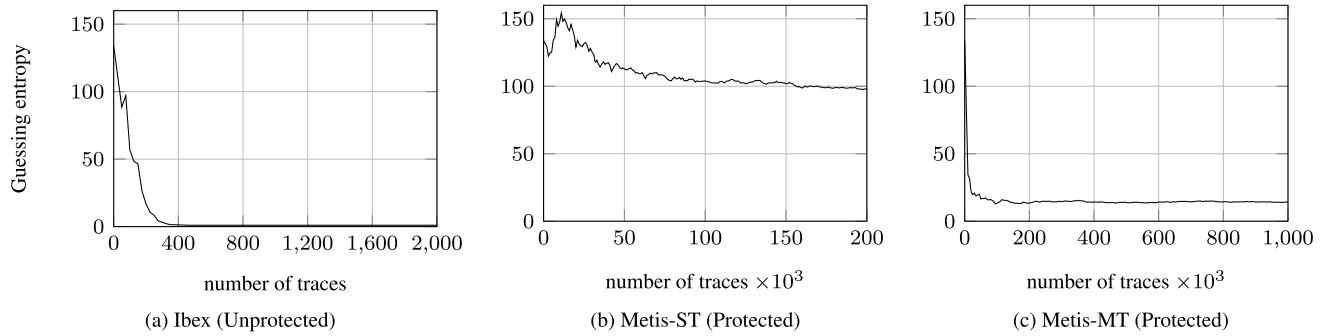


FIGURE 11. Results of computing the guessing entropy, employing as a ranking the outcome of a CPA attack against AES-128, targeting the computation of the SubBytes primitive on both the unprotected Ibex and on Metis. All the guessing entropies were computed on 30 independent experiments, each one performed with a different secret key.

as the fraction of the times the correct key guess is ranked by the attack methodology among the first o positions, over the number of attacks.

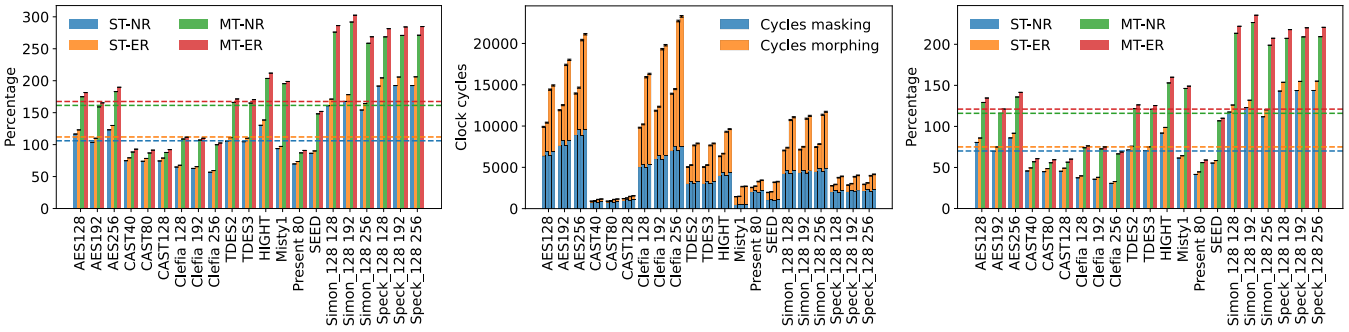
Figure 10 reports the results of the success rate over 30 attacks, performed employing the CPA ranking of the key candidates. The figure reports the trend of the fifth order success rate ($o = 5$), as a function of the number of traces employed to perform the attacks. As it can be seen, the fifth order success rate against the unprotected Ibex reaches 1, indicating that the correct key is always among the first five guesses, after 400 traces. By contrast, the attacks against both Metis implementations have a success rate which drops down to zero when more traces are added, indicating that a larger amount of information will not be useful to breach the security of the approach.

The *guessing entropy* metric was proposed in order to concretely estimate the number of remaining key guesses an attacker should perform, assuming that he is employing the information coming from a side channel attack as a support to the exhaustive search for the correct cipher key. The computation of the guessing entropy assumes that the attacker will employ the output of a ranking side channel attack of the same kind as the one considered by the success rate metric, as a prompt to start guessing the possible values of the key portions. As a consequence, the guessing entropy considers what is the average position of the correct key in the ranking provided by the side channel attack employed to compute it. Indeed, if an attacker conduces the exhaustive key search, enumerating each portion of the key according to the results of the side channel attack which he has lead, having the correct key ranked far from the first place will slow him down in his exhaustive search.

Figure 11 reports the results of computing the guessing entropy considering 30 ranking obtained from 30 independent CPA attacks. As it can be seen, the rank of the correct key in the unprotected Ibex SoC drops to 1, i.e., the correct key is ranked as the first to be guessed, after ≈380 traces. By contrast, the guessing entropy of the Metis protected implementation never drops to 1, indicating a persistent remaining guesswork to be made by the attacker. In particular, we recall that, since the side channel attack performed is attempting at extracting 8 key bits out of 128, $\frac{128}{8} = 16$ independent guessing attempts are required to derive the entire AES key. This in turn implies that a guessing entropy score of 32 results in $32^{16} = 2^{80}$ possible key guesses.

C. COMPUTATION PERFORMANCE EVALUATION

To provide a fair evaluation of the integrated code morphing overhead on a practically relevant workload, we evaluated Metis on all the available ISO/IEC standard symmetric block



(a) Overhead measured on execution time (b) Number of clock cycles spent in morphing and masking (c) Overhead measured on wall clock

FIGURE 12. Performance evaluation of the Metis SoC, comparing the results obtained on the ST and MT tiles, NR and ER refreshing strategies to the performance of the Ibex SoC. All dashed lines represent the geometric means of the results in the corresponding plot. Subfigure (a) reports the overheads in execution time taking into account the difference in the clock rate of the two SoCs: OpenTitan (48 MHz), Metis (40 MHz). Subfigure (b) reports the number of clock cycles spent in the morphing and masking computations by Metis. Subfigure (c) measures the overhead of Metis with respect to the plain OpenTitan when considering the wall clock.

ciphers, namely: ISO/IEC 18033-3 [29] wide-block ciphers AES and SEED, and narrow-block ciphers: Triple DES-EDE, CAST, HIGHT and Misty1; ISO/IEC 29192-2 [32] lightweight ciphers: Clefia and Present; ISO/IEC 29167-21 Speck [31] and ISO/IEC 29167-22 Simon [30] lightweight ciphers. We adapted the existing code bases for the execution on Metis enlarging the size of the variables requiring masking. While we performed the operation manually, we note that this modification was easily integrated in the compiler backend [1], as all the required information are available at compile time. We also annotated code to unroll the cipher round loops, as it is customary, given the small code size of block ciphers [2], [11]. We enable the code morphing in Metis through setting the morphing enable bit in the mstatus status register, before the start of the cipher execution and we measured performance with two different mask refreshing probabilities: 0.25 for Normal Refreshing (NR), and 0.75 for Extended Refreshing (ER). While Metis provides already a significant security margin on the MTD metric with NR, we provide its performance figures with ER to quantify the small additional overhead of ER.

Performance measurements are obtained adding Hardware Performance Monitor (HPM) counters to report the overall amount of instructions substituted, and the cycles spent in morphing or masking execution mode. Given the variable length of the executed tiles, the average performance over $1,000$ cipher executions is reported.

Figure 12 reports the results of the benchmark campaign in terms of execution time overhead of the symmetric ciphers with respect to an unmodified SoC. The overhead is reported both with respect to a comparison on the time taken (Figure 12a) taking into account the difference in clock frequency between Metis and Ibex, and the wall clock time taken (Figure 12c). Figure 12b reports the amount of overhead clock cycles, split by their task: executing morphed code or performing masked load-store operations with refreshing. The obtained results show how Metis achieves an overhead of $0.7\times$ when employing ST, and of $1.15\times$ when employing MT tiles and a refreshing probability of 0.25 (Normal Refreshing, NR). Boosting the mask refresh probability to 0.75 (Extended Refreshing, ER) only increases the overhead by an additional 5%, showing the advantage of performing integrated mask refreshing. The overall execution times of the ISO standard cipher suite have an overhead of $1.05\times$ when considering ST, and $1.61\times$ when considering MT, and a mask refresh probability of 0.25, retaining the similarly small extra overhead ($\approx 6\%$) for a mask refreshing with probability 0.75.

We now analyze the differences in the computational overheads of the ciphers, highlighting which block cipher design criteria are the most efficient when considering morphing-friendly block ciphers. Block ciphers relying on integer arithmetics as a means to achieve Boolean nonlinear functions, such as the ISO/IEC and Canadian standard CAST, and the South Korean standard SEED, report the lowest overhead among all ISO standard ciphers. This fact is a consequence of the lack of LUT-based S-Boxes, which in turn remove the need to perform LUT masking during the morphing of load operations. A particular point sets apart Speck and Simon, which, despite the use of non-LUT based nonlinear components, are characterized by a significant amount of bitwise rotations, for which dedicated tiless are present in the Metis morphing engine. The morphing of the rotations, together with the one of the bitwise xor makes the overwhelming majority of the Speck and Simon operations subject to morphing. Nonetheless the overall overhead experienced by Speck and Simon is at most $\approx 2.25\times$, which still compares favourably with the one-order-of-magnitude of pure masking overheads [49]. A similarly high overhead affects HIGHT, originating by the use of the same set of ISA operations, another South Korean standard cipher present in the ISO standard block cipher suite. Block ciphers employing tabulated LUTs to compute their nonlinear S-Box functions, i.e. AES, Clefia, Misty1 and Present, exhibit overheads in

line with the geometric mean of the figures, albeit half of the overhead or more is indeed caused by the additional load-store operations, as reported in Figure 12b. The performance figures obtained on AES, arguably the most widely employed block cipher, match closely the geometric mean of the benchmark, thus providing a good representative for cross-evaluation with other morphing solutions.

TABLE 2. Comparison among code morphing approaches.

Technique	Runs w/o morphing	Execution time (geo. mean) AES-SBox	ISO-STD	[11]	Code size
Metis ST-NR	1	2.14×	2.10×	2.21×	1.00×
Metis ST-ER	1	2.21×	2.16×	2.28×	1.00×
Metis MT-NR	1	2.72×	2.67×	2.93×	1.00×
Metis MT-ER	1	2.79×	2.73×	3.00×	1.00×
[2]	1	396×	-	-	-
MEET [4]	1	-	6.7×	-	5.18×
Odo High [11]	1	-	-	175×	3.66×
Odo Low [11]	1	-	-	67.4×	1.65×
[2]	100	5.33×	-	-	-
Odo High [11]	100	-	-	6.7×	3.66×
Odo Low [11]	100	-	-	4.2×	1.65×

A comparison of the performance of Metis with the one achieved by the existing code morphing approaches is reported in Table 2. The software code morphing approaches in [2], [11] propose to perform a periodic dynamic recompilation of the block cipher code once every r full block cipher executions. This allows to amortize the cost of code morphing over multiple executions of the cipher, at the cost of a decreased protection effectiveness. In particular, executing the cipher a number of times equal to the MTD without morphing the code maskes it vulnerable to a vanilla CPA attack [2]. The alternative approach employed in [4] reduces the cost of dynamic recompilation, compiling ahead of time multiple code variants. While this removes the cost of online recompilation, it limits the flexibility of the approach and is characterized by a code size blowup of about 5×. We note that our approach continuously performs code morphing, without any code size overhead.

We compare the execution time of Metis against the software based code morphing approaches on the set of block ciphers employed by each of the approaches (AES only for [2], the set of ISO/IEC standard ciphers save for Simon and Speck for [4] and the subset of ISO/IEC standard ciphers tackled in [11]). Comparing Metis with MT-ER morphing against continuously morphing approaches yields a 141× improvement w.r.t the original [2] morphing approach, a 21× to 58× improvement against [11], depending on the set of code morphing techniques selected in [11], and a 2.4× improvement on [4], paired with a null overhead on the code size w.r.t. the quintuplication operated by [4]. Finally, comparing Metis against the code-morphing approaches of [2], [11], considering the case of a code morphing action every 100 block cipher executions (i.e., a recompilation every $\approx 10^5$ instructions) we report a still significant improvement, namely 1.91× for [2] and 1.4×–2.23× with respect to [11].

To put the obtained overheads in perspective, it is useful to compare them with the ones of protections relying on masking schemes: a typical first-order masking countermeasure, with a two share encoding, is responsible for an increased clock cycles requirements of 10x on average, and also as much as 129x in case of the AES cipher [49].

V. CONCLUSION

We proposed Metis, a compact CPU design integrating a code morphing engine and transparent support for masking of memory operations and randomized mask refreshing. Our solution provides security improvements in the MTD metric up to 1250×, while keeping the increase in logic resources down to 25% of the compact CPU (roughly in the ARM Cortex-M3 class) we built on, and down to ≈5% of the original OpenTitan SoC. The computing performance overhead of Metis improves on software based solutions doing continuous runtime code morphing by 21× to 141×, and by 2.4× with respect to a solution performing static morphing and randomized execution, with an additional 5× code overhead reduction with respect to the latter.

REFERENCES

[1] G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi, "Compiler-based side channel vulnerability analysis and optimized countermeasures application," in *Proc. 50th Annu. Design Autom. Conf. (DAC)*, Austin, TX, USA, May/Jun. 2013, pp. 81:1–81:6, doi: 10.1145/2463209.2488833.

[2] G. Agosta, A. Barenghi, and G. Pelosi, "A code morphing methodology to automate power analysis countermeasures," in *Proc. 49th Annu. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Jun. 2012, pp. 77–82, doi: 10.1145/2228360.2228376.

[3] G. Agosta, A. Barenghi, and G. Pelosi, "Compiler-based techniques to secure cryptographic embedded software against side-channel attacks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 8, pp. 1550–1554, Aug. 2020, doi: 10.1109/TCAD.2019.2912924.

[4] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "The MEET approach: Securing cryptographic embedded software against side channel attacks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 8, pp. 1320–1333, Aug. 2015, doi: 10.1109/TCAD.2015.2430320.

[5] A. Althoff, J. McMahan, L. Vega, S. Davidson, T. Sherwood, M. B. Taylor, and R. Kastner, "Hiding intermittent information leakage with architectural support for blinking," in *Proc. 45th ACM/IEEE Annu. Int. Symp. Comput. Archit. (ISCA)*, Los Angeles, CA, USA, Jun. 2018, pp. 638–649, doi: 10.1109/ISCA.2018.00059.

[6] M. K. F. Arsath, V. Ganesan, R. Bodduna, and C. Rebeiro, "PARAM: A microprocessor hardened for power side-channel attack resistance," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, San Jose, CA, USA, Dec. 2020, pp. 23–34, doi: 10.1109/HOST45689.2020.9300263.

[7] J. Balasch, B. Gierlichs, O. Reparaz, and I. Verbauwhede, "DPA, bitslicing and masking at 1 GHz," in *Cryptographic Hardware and Embedded Systems—CHES* (Lecture Notes in Computer Science), vol. 9293, T. Güneysu and H. Handschuh, Eds. Berlin, Germany: Springer, Sep. 2015, pp. 599–619, doi: 10.1007/978-3-662-48324-4_30.

[8] A. Barenghi, W. Fornaciari, G. Pelosi, and D. Zoni, "Scramble suit: A profile differentiation countermeasure to prevent template attacks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 9, pp. 1778–1791, Sep. 2020, doi: 10.1109/TCAD.2019.2926389.

[9] A. Barenghi and G. Pelosi, "Side-channel security of superscalar CPUs: Evaluating the impact of micro-architectural features," in *Proc. 55th Annu. Design Autom. Conf.*, San Francisco, CA, USA, Jun. 2018, pp. 120:1–120:6, doi: 10.1145/3195970.3196112.

[10] A. G. Bayrak, N. Velickovic, P. Ienne, and W. P. Burleson, "An architecture-independent instruction shuffler to protect against side-channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 20:1–20:19, 2012, doi: 10.1145/2086696.2086699.

[11] N. Belleville, D. Couroussé, K. Heydemann, and H. Charles, "Automated software protection for the masses against side-channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, p. 47:1–47:27, 2019, doi: 10.1145/3281662.

[12] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems—CHES* (Lecture Notes in Computer Science), vol. 3156, M. Joye and J. Quisquater, Eds. Berlin, Germany: Springer, Aug. 2004, pp. 16–29, doi: 10.1007/978-3-540-28632-5_2.

[13] F. Bruguier, P. Benoit, L. Torres, L. Barthe, M. Bourree, and V. Lomne, "Cost-effective design strategies for securing embedded processors," *IEEE Trans. Emerg. Topics Comput.*, vol. 4, no. 1, pp. 60–72, Jan. 2016, doi: 10.1109/TETC.2015.2407832.

[14] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *Cryptographic Hardware and Embedded Systems—CHES* (Lecture Notes in Computer Science), vol. 2523, B. S. K. Jr., Ç. K. Koç, and C. Paar, Eds. Berlin, Germany: Springer, Aug. 2002, pp. 13–28, doi: 10.1007/3-540-36400-5_3.

[15] D. Couroussé, T. Barry, B. Robisson, N. Belleville, P. Jaillon, O. Potin, H. L. Bouder, J. Lanet, and K. Heydemann, "All paths lead to Rome: Polymorphic runtime code generation for embedded systems," in *Proc. 5th Workshop Cryptogr. Secur. Comput. Syst. (CS2)* 2018, Manchester, U.K., Jan. 2018, pp. 17–18, doi: 10.1145/3178291.3178296.

[16] D. Couroussé, T. Barry, B. Robisson, P. Jaillon, O. Potin, and J. Lanet, "Runtime code polymorphism as a protection against side channel attacks," in *Information Security Theory and Practice*, (Lecture Notes in Computer Science), vol. 9895, S. Foresti and J. López, Eds. Cham, Switzerland: Springer, Sep. 2016, pp. 136–152, doi: 10.1007/978-3-319-45931-8_9.

[17] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity," in *Proc. 22nd Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, Feb. 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/thwarting-cache-side-channel- attacks-through-dynamic-software-diversity/

[18] D. Das, J. Danial, A. Golder, N. Modak, S. Maity, B. Chatterjee, D. Seo, M. Chang, A. Varna, H. Krishnamurthy, S. Mathew, S. Ghosh, A. Raychowdhury, and S. Sen, "27.3 EM and power SCA-resilient AES-256 in 65 nm CMOS through 350× current-domain signature attenuation," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, San Francisco, CA, USA, Feb. 2020, pp. 424–426, doi: 10.1109/ISSCC19947.2020.9062997.

[19] D. Das, S. Maity, S. B. Nasir, S. Ghosh, A. Raychowdhury, and S. Sen, "High efficiency power side-channel attack immunity using noise injection in attenuated signature domain," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, McLean, VA, USA, May 2017. pp. 62–67, doi: 10.1109/HST.2017.7951799.

[20] F. De Santis, T. Bauer, and G. Sigl, "Hiding higher-order univariate leakages by shuffling polynomial masking schemes: A more efficient, shuffled, and higher-order masked AES S-box," in *Proc. ACM Workshop Theory Implement. Secur. (TISCCS)*, Vienna, Austria, Oct. 2016, pp. 17–26, doi: 10.1145/2996366.2996370.

[21] T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. T. M. Shalmani, "On the power of power analysis in the real world: A complete break of the KeeLoq code hopping scheme," in *Advances in Cryptology—CRYPTO* (Lecture Notes in Computer Science), vol. 5157, D. A. Wagner, Ed. Berlin, Germany: Springer, Aug. 2008, pp. 203–220, doi: 10.1007/978-3-540-85174-5_12.

[22] Embedded Microprocessor Benchmark Consortium. (2020). *Coremark Benchmark Suite*. [Online]. Available: https://www.eembc.org/coremark/download.php

[23] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco, S. Malik, M. Tiwari, and T. M. Austin, "Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, Providence, RI, USA, Apr. 2019, pp. 469–484, doi: 10.1145/3297858.3304037.

[24] D. Genkin, L. Pachmanov, I. Pipman, A. Shamir, and E. Tromer, "Physical key extraction attacks on PCs," *Commun. ACM*, vol. 59, no. 6, pp. 70–79, May 2016, doi: 10.1145/2851486.

[25] D. Genkin, A. Shamir, and E. Tromer, "Acoustic cryptanalysis," *J. Cryptol.*, vol. 30, no. 2, pp. 392–443, Apr. 2017, doi: 10.1007/s00145-015-9224-2.

[26] H. Groß, M. Jelinek, S. Mangard, T. Unterluggauer, and M. Werner, "Concealing secrets in embedded processors designs," in *Smart Card Research and Advanced Applications* (Lecture Notes in Computer Science), vol. 10146, K. Lemke-Rust and M. Tunstall, Eds. Cham, Switzerland: Springer, Nov. 2016, pp. 89–104, doi: 10.1007/978-3-319-54669-8_6.

[27] A. Heuser, O. Rioul, and S. Guilley, "Good is not good enough—Deriving optimal distinguishers from communication theory," in *Cryptographic Hardware and Embedded Systems—CHES* (Lecture Notes in Computer Science), vol. 8731, L. Batina and M. Robshaw, Eds. Springer, Sep. 2014, pp. 55–74, doi: 10.1007/978-3-662-44709-3_4.

[28] G. Hospodar, B. Gierlichs, E. De Mulder, I. Verbauwhede, and J. Vandewalle, "Machine learning in side-channel analysis: A first study," *J. Cryptograph. Eng.*, vol. 1, no. 4, pp. 293–302, Dec. 2011, doi: 10.1007/s13389-011-0023-x.

[29] *Information Technology—Security Techniques—Encryption Algorithms—Part 3: Block Ciphers*, Standard ISO/IEC 18033-3:2010, 2010. [Online]. Available: https://www.iso.org/standard/51582.html

[30] *Information Technology—Automatic Identification and Data Capture Techniques—Part 21: Crypto Suite SIMON Security Services for Air Interface Communications*, Standard ISO/IEC 29167-21:2018, 2018. [Online]. Available: https://www.iso.org/standard/70388.html

[31] *Information Technology—Automatic Identification and Data Capture Techniques—Part 22: Crypto Suite SPECK Security Services for Air Interface Communications*, Standard ISO/IEC 29167-22:2018, 2018. [Online]. Available: https://www.iso.org/standard/70389.html

[32] *Information Security—Lightweight Cryptography—Part 2: Block Ciphers*, Standard ISO/IEC 29192-2:2019, 2019. [Online]. Available: https://www.iso.org/standard/78477.html

[33] J. Irwin, D. Page, and N. P. Smart, "Instruction Stream Mutation for Non-Deterministic Processors," in *Proc. 13th IEEE Int. Conf. Appl.-Specific Syst., Archit., Processors (ASAP)*, San Jose, CA, USA, Jul. 2002, pp. 286–295, doi: 10.1109/ASAP.2002.1030727.

[34] Y. Ishai, A. Sahai, and D. A. Wagner, "Private circuits: Securing hardware against probing attacks," in *Advances in Cryptology—CRYPTO* (Lecture Notes in Computer Science), vol. 2729, D. Boneh, Ed. Berlin, Germany: Springer, Aug. 2003, pp. 463–481, doi: 10.1007/978-3-540-45146-4_27.

[35] P. Kiaei, D. Mercadier, P.-E. Dagand, K. Heydemann, and P. Schaumont, "Custom instruction support for modular defense against side-channel and fault attacks," in *Proc. Cryptol. ePrint Arch., Rep.*, 2020, pp. 1–34. [Online]. Available: https://eprint.iacr.org/2020/466

[36] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, May 2019, pp. 1–19, doi: 10.1109/SP.2019.00002.

[37] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology—CRYPTO* (Lecture Notes in Computer Science), vol. 1666, M. J. Wiener, Ed. Berlin, Germany: Springer, Aug. 1999, pp. 388–397, doi: 10.1007/3-540-48405-1_25.

[38] E. Laohavaleeson and C. Patel, "Current flattening circuit for DPA countermeasure," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, Anaheim, CA, USA, Jun. 2010, pp. 118–123, doi: 10.1109/HST.2010.5513104.

[39] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proc. 27th USENIX Secur. Symp. Secur. (USENIX)*, Baltimore, MD, USA, Aug. 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[40] *Ibex Documentation*, lowRISC, Cambridge, U.K., Jan. 2020.

[41] lowRISC Community Interest Company. (Jan. 2020). *Ibex RISC-V Core*. [Online]. Available: https://github.com/lowRISC/ibex

[42] lowRISC Community Interest Company. (Jan. 2020). *OpenTitan Documentation*. [Online]. Available: https://docs.opentitan.org/

[43] lowRISC Community Interest Company. (Jan. 2020). *OpenTitan Github Repository*. [Online]. Available: https://github.com/lowRISC/opentitan

[44] T. S. Messerges, "Using second-order power analysis to attack DPA resistant software," in *Cryptographic Hardware and Embedded Systems—CHES* (Lecture Notes in Computer Science), vol. 1965, Ç. K. Koç and C. Paar, Eds. Berlin, Germany: Springer, Aug. 2000, pp. 238–251, doi: 10.1007/3-540-44499-8_19.

[45] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, F. Piessens, and D. Gruss, "Plundervolt: How a little bit of undervolting can create a lot of trouble," *IEEE Secur. Privacy*, vol. 18, no. 5, pp. 28–37, Sep. 2020, doi: 10.1109/MSEC.2020.2990495.

[46] D. Oswald and C. Paar, "Breaking mifare DESFire MF3ICD40: Power analysis and templates in the real world," in *Cryptographic Hardware and Embedded Systems—CHES*, (Lecture Notes in Computer Science), vol. 6917, B. Preneel and T. Takagi, Eds. Berlin, Germany: Springer, Sep./Oct. 2011, pp. 207–222. 10.1007/978-3-642-23951-9_14.

[47] D. Oswald, B. Richter, and C. Paar, "Side-channel attacks on the Yubikey 2 one-time password generator," in *Research in Attacks, Intrusions, and Defenses* (Lecture Notes in Computer Science), vol. 8145, S. J. Stolfo, A. Stavrou, and C. V. Wright, Eds. Berlin, Germany: Springer, Oct. 2013, pp. 204–222, doi: 10.1007/978-3-642-41284-4_11.

[48] A. Peetermans, V. Rozic, and I. Verbauwhede, "A highly-portable true random number generator based on coherent sampling," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Barcelona, Spain, Sep. 2019, pp. 218–224, doi: 10.1109/FPL.2019.00041.

[49] M. Rivain and E. Prouff, "Provably secure higher-order masking of AES," in *Cryptographic Hardware and Embedded Systems, CHES*, S. Mangard and F.-X. Standaert, Eds. Berlin, Germany: Springer, 2010, pp. 413–427.

[50] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn, "IoT goes nuclear: Creating a ZigBee chain reaction," *IEEE Secur. Privacy*, vol. 16, no. 1, pp. 54–62, Jan. 2018, doi: 10.1109/MSP.2018.1331033.

[51] S. A. Seyyedi, M. Kamal, H. Noori, and S. Safari, "Securing embedded processors against power analysis based side channel attacks using reconfigurable architecture," in *Proc. IEEE/IFIP 9th Int. Conf. Embedded Ubiquitous Comput. (EUC)* 2011, Melbourne, VIC, Australia, Oct. 2011, pp. 255–260, doi: 10.1109/EUC.2011.62.

[52] F. Standaert, T. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," *IACR Cryptol. ePrint Arch.*, vol. 2006, p. 139, Jun. 2009. [Online]. Available: http://eprint.iacr.org/2006/139

[53] STMicroelectronics. (2020). *STM32F407/417 Cortex-M4 Based Microcontroller Datasheet*. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32f415rg.pdf

[54] M. Tunstall, C. Whitnall, and E. Oswald, "Masking tables—An underestimated security risk," in *Fast Software Encryption* (Lecture Notes in Computer Science), vol. 8424, S. Moriai, Ed. Singapore: Springer, Mar. 2013, pp. 425–444, doi: 10.1007/978-3-662-43933-3_22.

FRANCESCO ANTOGNAZZA is currently pursuing the Ph.D. degree with the Politecnico di Milano, Italy. His research interest includes efficient and side channel-resistant hardware implementations of cryptographic primitives, with a focus on post-quantum asymmetric cryptosystems.

ALESSANDRO BARENGHI is currently an Associate Professor with the Politecnico di Milano, Italy. He has published more than 80 articles in international peer-reviewed venues. His research interests include computer and network security, formal languages, and compilers.

GERARDO PELOSI (Member, IEEE) is currently an Associate Professor with the Politecnico di Milano, Italy. He is a co-inventor of ten patents concerning the design of cryptographic systems. He has published more than 90 articles in international peer-reviewed journals and conference proceedings. His main research interests include computer security, cryptography, security in hardware and in the area of data security, and privacy.