# SKIVA: Flexible and Modular Side-channel and Fault Countermeasures

Pantea Kiaei[1], Darius Mercadier[2], Pierre-Evariste Dagand[3], Karine Heydemann[4] and Patrick Schaumont[5]

[1] Virginia Tech, Blacksburg, USA, pantea95@vt.edu
[2] LIP6, Paris, France, darius.mercadier@gmail.com
[3] LIP6, Paris, France, pierre-evariste.dagand@lip6.fr
[4] LIP6, Paris, France, karine.heydemann@lip6.fr
[5] Virginia Tech, Blacksburg, USA, schaum@vt.edu

**Abstract.** We describe SKIVA, a customized 32-bit processor enabling the design of software countermeasures for a broad range of implementation attacks covering fault injection and side-channel analysis of timing-based and power-based leakage. We design the countermeasures as variants of bitslice programming. Our protection scheme is flexible and modular, allowing us to combine higher-order masking – fending off side-channel analysis – with complementary spatial and temporal redundancy – protecting against fault injection. Multiple configurations of side-channel and fault protection enable the programmer to select the desired number of shares and the desired redundancy level for each slice. Recurring and security-sensitive operations are supported in hardware through a custom instruction set extension. The new instructions support bitslicing, secret-share generation, redundant logic computation, and fault detection. We demonstrate and analyze multiple versions of AES from a side-channel analysis and a fault-injection perspective, in addition to providing a detailed performance evaluation of the protected designs.

**Keywords:** Bitslicing · Side-channel attacks · Fault attacks · Custom-instruction extensions · Software Countermeasures

## 1 Introduction

Side-channel analysis and fault attacks have plagued cryptographic software on embedded processors for many years. The threat of power-based and timing-based side-channel leakage is well understood and countermeasures such as masking and constant-time programming figure prominently in the cryptographer's toolbox [RBV17, BDF+17]. In parallel, the research community has gained more insight into the fault behavior of hardware and software, thus greatly increasing the potency of fault attacks [YSW18, RNR+15]. The impact of fault attacks is minimized with fault detection and temporal or spatial redundancy of the software execution [LHB14, BCR16].

Although there exists an extensive array of specific, dedicated countermeasures, there is surprisingly few work available [RDB+18, SMG16, SBD+18] offering protection against both side-channel analysis *and* fault injection. This is especially true for software. The programmer is left selecting candidate solutions, figuring out if and how they can safely be assembled. This is not an easy task because countermeasures may interact in non-trivial (and unsafe) manners. For example, time-redundant software [CPT17] or error-detecting codes [RBIK12] as fault countermeasures may increase side-channel leakage, while constant-time programming as a side-channel countermeasure increases the risk of precisely synchronized fault attacks [SMKLM02].

In this paper, we introduce SKIVA, a processor that enables a modular approach to countermeasure design, giving programmers the flexibility to protect their ciphers against timing-based side-channel analysis, power-based side-channel analysis and/or fault injection at various levels of security. We leverage existing techniques in higher-order masking, in spatial and in temporal redundancy. Modularity is achieved through bitslicing, each countermeasure being expressed as a transformation from a bitsliced design into another bitsliced design. The capabilities of SKIVA are demonstrated on the Advanced Encryption Standard, but the proposed techniques can be applied to other ciphers as well.

**Tackling physical effects with software.** The protection of software against side-channel analysis and fault attacks is challenging. Side-channel leakage and faults are physical effects in the processor hardware. The programmer can control macro-level properties such as the control path of software or the amount and nature of memory references. But a large portion of software execution occurs "under the hood". As a processor fetches, decodes and executes each instruction, the sensitive data handled by an instruction moves through the processor hardware. The timing, power consumption, and fault sensitivity of each instruction is obscured to the programmer. Due to this hardware abstraction, predicting physical execution properties of sensitive data is very hard for the programmer, as the following examples illustrate:

- The instruction timing is determined by the micro-architecture pipeline configuration, the cache organization, the presence of branch prediction, among other factors. A programmer cannot predict the execution time of a program from the source code alone. The processor hardware is optimized to make the common case fast [PH12], but it is unable to deliver strong guarantees on the timing of a single, specific instruction. Instead, instruction timing is strongly affected by the execution context.
- The instruction power dissipation is affected by signal transitions on programmer-invisible processor structures such as buses, buffers, memories, and logic. The power dissipation of these signal transitions is proportional to the Hamming distance between former and current data values in the hardware. In many cases, for example when the hardware is a shared resource, the former data value is unknown or invisible to the programmer.
- The instruction fault-sensitivity is determined by the electrical properties of hardware structures in the processor including their critical path and their threshold levels [BGV11, YSW18]. However, published hardware datasheets only list typical, maximum or minimum ratings. The fault sensitivity of a specific instruction is therefore unknown to the programmer. This applies not only to timing, but also to power dissipation.

As a result, contemporary processors do not offer a comprehensive guarantee on the physical execution properties of hardware: implementing a secure (yet reasonably efficient) cipher is thus exceedingly hard [BGG$^+$14].

**Symmetry from bitslicing.** In order to get a handle on this problem, we adopt a bitsliced execution model. In the bitsliced model, the $n$-bit datapath of the processor is seen as $n$ 1-bit processors operating in parallel. Such an SIMD array of $n$ parallel 1-bit processors offers a significant degree of symmetry and regularity. Through this symmetry, the programmer gets a grip on the physical execution properties of software, at least in a relative sense:

- The cycle-time of parallel bitslices within an instruction is matched. The amount of clock cycles, used by any single bitslice within a processor word to complete a bitslice program, is the same as for any other bitslice of the same word. This property also holds under typical processor latency effects (pipeline hazards, caches, branch prediction). Every bit of a processor word experiences the same clock-cycle delay.

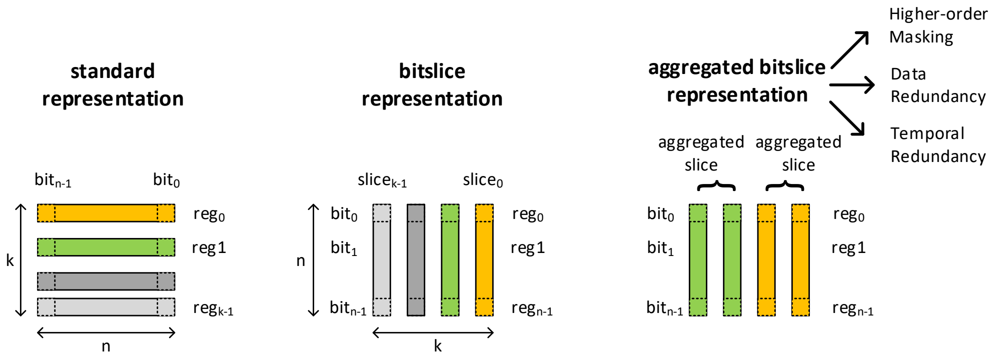


**Figure 1:** In a standard representation, processor registers are allocated per data word. In a bitsliced representation, processor registers are allocated per bit-weight of a block of data words. In an aggregated bitslice representation, multiple bitslices are allocated per data bit. Aggregated bitslices can be shares of a masked design, redundant data of a fault-protected design, or a combination of those.

- The power consumption of parallel bitslices in an instruction is matched. If two parallel bits in a CPU word make the same transition under the same bit-wise instruction, then they will have the same power dissipation. Of course, process manufacturing variations, and variations in on-chip and PCB routing may cause small differences in power. But these are second-order effects compared to the first-order symmetry obtained by the bitslices within a processor word.

- The instruction fault sensitivity of parallel bitslices in an instruction is matched. For example, if two parallel bits in a CPU word experience the same timing fault under the same bit-wise instruction, then we expect a matched fault effect. As with power consumption, there may be small static variations due to process manufacturing and routing [LSG$^+$10].

**Countermeasure design through bitslice aggregation.** The symmetry of bitslices in a processor word is the basis for the modular protection schemes enabled by SKIVA. Figure 1 demonstrates three different organizations of a register file in a processor. We obtain the bitslice representation through a matrix transposition of the input data, so that one processor register contains all bits of a given weight. The key idea of bitslice aggregation is to allocate multiple slices to the representation of each data-bit.

*Timing-based Side-channel Leakage.* Bitslices – aggregated or not – are naturally synchronized, and always use a matching amount of cycles. Furthermore, bitslice programming naturally leads to straight-line programs with precisely defined timing characteristics [Bih97]. Straight-line programs have to be written at the level of bit-operations, and hence they are not easy to develop for the programmer. Fortunately, bitslice software generation can be automated [Por01, SS16, MDLM18].

*Power-based Side-channel Leakage.* In an order-$d$ masked implementation, a single secret data bit is split into $d + 1$ shares using a masking method and $d$ random shares. An order-$d$ masked implementation is theoretically protected against order-$d$ side-channel attacks. By allocating different shares as parallel slices, we obtain a parallel masked implementation [BDF$^+$17], as demonstrated by Balasch *et al.* for ARM [BGRV15] and by Gregoire *et al.* for ARM-NEON [GPSS18]. Conceptually, their aggregate represents a single bit.

*Fault redundancy.* Bitslice aggregation enables spatial redundancy by encoding a single bit multiple times in each of the available slices. This allows the detection of data errors such as bit-flip and stuck-at faults and it is one element in a comprehensive fault countermeasure [PYGS16, LCFS18]. Interestingly, aggregation of bitslices can also offer support for temporal redundancy. In an iterative block cipher, for example, different bitslices can execute different rounds of a redundant cipher. This protects against instruction-skip and faults on the control path.

**Contributions.** In this paper we introduce SKIVA, a processor with built-in support for modular countermeasures against side-channel analysis and fault analysis. We make the following contributions:

1. A flexible and modular methodology for designing countermeasures. It enables the combination of a higher-order masking with spatial fault-redundancy and with temporal fault-redundancy. The number of shares and fault-redundancy levels are statically determined by the programmer (single, double, quadruple shares and single, double, quadruple fault-redundancy).
2. Hardware support for the proposed methodology in SKIVA, a processor with instruction set extensions specialized for bitsliced transposition, bitsliced masked operation, bitsliced fault detection, redundant bitsliced expansion and Boolean operations on complementary data.
3. Performance analysis of the Advanced Encryption Standard on SKIVA, under multiple levels of side-channel and fault-resistance.
4. Side-channel leakage evaluation of SKIVA implemented as a soft-core processor on a SAKURA-G FPGA board, extensive code size and performance evaluation, theoretical as well as empirical analysis of fault detection coverage.

**Outline.** In Section 2, we capture preliminaries to establish a common background among readers. We discuss the attacker model (Section 2.1) and review the related work, covering the design of countermeasures (Section 2.2) and the design of bitsliced software (Section 2.3). In Section 3, we introduce several modular countermeasure schemes. Starting with bitslicing (Section 3.1), we describe our systematic treatment of higher-order masking (Section 3.2), intra-instruction redundancy (Section 3.3), and temporal redundancy (Section 3.4). Finally, we demonstrate that these features naturally combine to form a coherent countermeasure within the assumptions of the attacker model (Section 3.5). In Section 4, we discuss the design and implementation of SKIVA, covering the instruction set extension and its overhead. In Section 5, we evaluate the software performance results, quantify the side-channel leakage of several levels of countermeasures, analytically and empirically bound the impact of fault attacks.

## 2 Preliminaries

We introduce the attacker model that is covered by our countermeasures (Section 2.1). We then review the literature for existing protection schemes (Section 2.2), with an eye toward modular techniques as well as countermeasures against fault and side-channel attacks. Finally, we recall the notion of bitslicing and review related work protecting bitsliced designs against fault-attacks and side-channel attacks (Section 2.3).

### 2.1 Attacker Model

The attacker model captures the presumed capabilities of an attacker. SKIVA considers adversaries with fault-injection and side-channel measurement capabilities.

**Fault attacker model.** We consider an attacker who intends to perform Differential Fault Analysis (DFA) [BS97, TMA11] or Statistical Fault Analysis (SFA) [FJLT13]. To carry out such an attack, she must induce a fault on an intermediate value or on control flow (*e.g.*, to force a loop count reduction [DMM+13]). Fault injection is achieved by stressing the electrical environment of the digital hardware. This induces transient faults that set, reset, or flip one or several bits in a storage element (register or memory) of the platform. The exact effect of a fault (or its statistical distribution) depends highly on the injection technique, its parameters, the target architecture, the manufacturing technology of the attacked device and the attacker's skills and time.

It is however generally agreed that there is a trade-off between the temporal and spatial resolution of a fault on the one hand, and the complexity of the fault injection on the other [YSW18]. The presumed fault attacker of SKIVA is not all-powerful: we consider low-cost injection means, with limited temporal resolution and/or spatial control over the fault effects. Concretely, we assume that the attacker is able to inject transient faults affecting a selected bit, byte, half-word, or word. We further restrict multi-bits faults to either setting or resetting the entire byte, half-word, or word (*stuck-at 1* and *stuck-at 0* models), or overwriting a selected byte, half-word or word with a random value. This excludes expensive equipment (*e.g.*, multi-spot laser [Col19]) and the ability to inject chosen values.

At the software level, this manifests itself as either a data corruption or an instruction corruption. A data corruption results from either a direct corruption (*e.g.*, a corruption of data transfer, of the bus, data path, or computational logic) or from indirect corruption (*e.g.*, modification of an instruction opcode). Instruction corruption may occur either during instruction fetch [MDH+13] or instruction read from Flash [Col19]. Most instruction corruptions reduce to a data corruption: an instruction is substituted for another, leading to an incorrect value to be stored in a register. In line with the limited capabilities of our attacker, we consider that an attacker is unable to corrupt the address of a jump to a chosen value. We therefore model control faults as *instruction skip*, whereby a chosen instruction is simply ignored during execution.

**Side-channel attacker model.** SKIVA is oriented towards embedded applications. The attacker controls data input, and can perform chosen-plaintext or chosen-ciphertext operations. This enables the gamut of differential analysis techniques. We assume an attacker who can observe the timing as well as the power dissipation of the processor. Timing measurements, such as done by precise measurement of input/output operations, proceed at the cycle-accurate level. This resolution enables extraction of data-dependent control-flow, and a slew of micro-architectural attacks [GYCH16]. The attacker is also able to monitor power dissipation by shunting the power supply or by measuring electromagnetic emissions. SKIVA is thus subject to side-channel attacks such as cache-timing analysis [Ber05], correlation power analysis (CPA) [BCO04], and differential electromagnetic analysis [CPM+18]. We also consider high-order side-channel analysis. The use of aggregated bitslices in SKIVA means that all shares of a secret are processed in parallel. In this contribution, we aim to demonstrate that SKIVA successfully supports this mode of operation. For this reason, we use a univariate leakage assessment methodology [SM15a] that evaluates leakage in the sample stream at multiple different leakage orders. In a generalized higher-order side-channel evaluation methodology, the attacker would also combine independent observations of side-channel leakage. Multi-variate leakage evaluation is outside of the scope of this contribution.

**Combined attacker model.** We also consider the case of an *active* attacker, combining both side-channel measurements and fault injection [GM10, AVFM07]. For instance, it has been shown that fault protection mechanisms tend to increase leakage and thus facilitate side-channel analysis [RBIK12, CPT17]. We take this effect into account in our experimental evaluation (Section 5.2).

Faults can also be used to mitigate the effect of SCA countermeasures [RLK11, DV12]. Lacking a well-established methodology to evaluate countermeasures against such attacks, we exclude it from our attacker model. In particular, we assume that our target platform offers an embedded and protected True Random Number Generator (TRNG) providing 32 bit of randomness at regular intervals in a dedicated register. We shall therefore ignore fault attacks specifically targeting the TRNG to disable re-masking [YYP+18].

Recent advances in the area of Statistical Ineffective Fault Attacks (SIFA) [DEK+18] have demonstrated that countermeasures based (solely) on fault detection may be insufficient, even in the context of a masked implementation. SKIVA offers a framework for exploring the design space of countermeasures resilient to SIFA, such as self-destruction [YGD+16], fault-correction, or hiding. However, designing and evaluating countermeasures against SIFA is a burgeoning area of research: in the present work, we therefore exclude this vector from our attacker model.

## 2.2 Countermeasures

A limitation of many countermeasures is that they are tailored to a specific algorithm. A *systematic* countermeasure is one which can be applied to any cryptographic operation. Instruction-level countermeasures are generic. They can be applied at the assembly level or as part of a compiler pipeline. We strive to design a set of systematic countermeasures that are *modular*, *i.e.* program transformations that can safely be chained one after the other, yielding an overall protection equal to the sum of its parts. Armed with these building blocks, programmers can adjust the security of their ciphers at will.

**Systematic countermeasures.** Systematic fault countermeasures are possible through automated instruction duplication and control-flow tracking [PHBC17, LHB14] or by exploiting intra-instruction redundancy in the target instruction set [CSN+17]. Intra-instruction redundancy is enabled either by SIMD or custom instructions (Section 4).

Systematic side-channel countermeasures against power-based side-channel analysis have overwhelmingly used masking techniques, driven by correctness criteria for the resulting side-channel security such as Perfect Masking [BGK05], Threshold Implementations [NRR06], and DOM [GMK16]. The difficulty of producing secure and efficient masked implementations in software has lead to various attempts at automating this process. In the setting of the CAO domain-specific language [BMP+11], it has been shown that the (strictly necessary) masking gadgets can be automatically synthesized from user-given annotations identifying public and private data [MOPT11, MOPT12]. A similar technique can be applied directly to C code [ABMP13, EW14] or assembly [BRN+15], instrumenting the LLVM compiler infrastructure to carry public/private annotations to the intermediate representation, performing static analysis to identify vulnerable program points and inserting standard masking gadgets [RP10]. Threshold implementation lends itself naturally – by its very design – to a systematic treatment, which has been implemented as a compilation pass in the LLVM compiler [LAZ+17].

**Modular countermeasures.** To the best of our knowledge, very few work tackles the issue of systematically protecting a cipher against both faults and side-channel analysis. One line of work, targeting hardware implementations, applies error-detecting codes on top of a threshold implementation [SMG16]. Another approach is the "tile-probe-and-fault" model [RDB+18] that postulates the physical isolation of the underlying hardware (the tiles). However, from the author's own admission, "this model [. . . ] does not perfectly fit commercial off-the-shelf multi-core architectures". To address these shortcomings, the

FRIT permutation [SBD$^{+}$18] has been specifically designed to allow inexpensive fault detection while being protected against side-channel attacks. This latter work demonstrates that combined defense is feasible at a software level (by exhibiting a secure, bitsliced implementation of FRIT). However, supporting legacy ciphers remains an open question.

## 2.3 Bitsliced software design

Bitslicing is a folklore technique to produce high-throughput, constant-time software implementations of cryptographic primitives [Bih97, KS09]. A cipher is expressed as a Boolean circuit. The circuit is compiled into a straight-line program by leveling the circuit and translating each Boolean operation to a corresponding bitwise CPU instruction. Since the CPU manipulates registers of 32 bits, running the resulting program amounts to running 32 parallel instances of the original Boolean circuit.

**Bitslicing versus wordslicing.** In a block cipher, the state variables are $k$-bit wide. The bitsliced version of the cipher will store these $k$ bits in a transposed manner, such that register $i$ will contain the $i$-th bit of the state. This approach has been used for DES [Bih97] as well as for AES [RSD06]. However, one can also adopt wordslicing, which stores groups of $b$ bits out of a $k$-bit state per register. A wordsliced design requires $k/b$ registers, as opposed to $k$ registers for a bitsliced design. Wordsliced design has been demonstrated for AES [Kön08, KS09]. The choice between bitslicing and wordslicing has significant impact on the efficiency of the resulting design. The resulting code also changes significantly with the slicing strategy. The bitsliced implementation of AES has to juggle with 128 machine words while being restricted to straightforward logical instructions. The wordsliced implementation of AES fits within 8 registers, at the expense of complex permutations within individual words. On an embedded RISC-like CPU, our experiments have shown that the bitsliced implementation yields a higher throughput than the wordsliced one (Section 5.1). Conversely, on a high-end SIMD CPU, earlier work has shown that wordslicing is key to reach speed records in software encryption [KS09]. The lack of SIMD instructions and the lesser register pressure for RISC CPUs thus favors bitsliced implementations, hence our focus on bitslicing in the present work.

**Countermeasures for bitsliced designs.** Many hardware-oriented countermeasures can be applied as transformations on the Boolean programs of bitsliced designs. An early effort to address power-based side-channel leakage is the duplication method [DPA00]. More recently, several masking-oriented techniques have been proposed [CS18, BDF$^{+}$17, JS17, GPSS18]. Bitslicing is also a systematic countermeasure against timing attacks. By construction, a Boolean program runs in constant time. For instance, S-boxes have data-independent runtime when run as Boolean programs. Similarly, conditionals in a Boolean program are implemented through data-multiplexing: both results are (sequentially) computed and the relevant output is obtained by demultiplexing these intermediary results based on the conditional. Finally, the massively parallel nature of a bitsliced implementation can be exploited to provide intra-instruction redundancy (encrypting the same data in redundant slices) as well as various forms of temporal redundancy (processing data at distinct rounds in distinct, randomly-chosen slices) [PYGS16, LCFS18]. In a bitsliced setting, these techniques translate into an end-to-end protection, protecting a cipher from the moment the plain text is introduced to the moment the cipher text is produced.

In the following, we demonstrate that, with some hardware support, bitslicing provides a sound basis to generalize some of the systematic protection schemes presented in the literature and gain protection against both attacks.

## 3 Modular design of countermeasures

In this section, we present the four protection mechanisms we adopt – bitslicing to protect against timing attacks, higher-order masking to protect against side-channels, intra-instruction redundancy to protect against data faults and temporal redundancy to protect against control faults – and exhaustively explore this design space opened up by our ability to compose them together.

Throughout this paper, we focus solely on the AES cipher targeting our 32-bit SKIVA processor. We chose the AES cipher for pedagogical reasons. It provides a yardstick to judge our protection scheme: it is well known in the community at large, both in terms of side-channel analysis, fault attack vectors as well as performance. However, the panel of techniques is not restricted to this cipher nor this processor: they naturally generalize – in a systematic manner – to any cipher in sliced form, for processors of arbitrary width as well as design (RISC as well as CISC, SIMD or not). We leave it to future work to evaluate their effectiveness on a broader range of cryptographic primitives and hardware platforms.

### 3.1 Constant-time programming

Our implementation of AES is fully bitsliced: the 128-bit input of the cipher is represented with 128 variables. Since each variable stores 32 bits on SKIVA, a single run of our primitive computes 32 parallel instances of AES. In Section 5.1, we show that, despite its register pressure, this implementation is the most efficient one on this RISC processor offering 32 general-purpose registers.

This bitsliced implementation of AES is the cornerstone of our work. The protection mechanisms presented in the following assume the availability of a bitsliced design while themselves producing a bitsliced design (of lesser parallelism) in return. The modularity of our approach lies in this simple observation: as long as there is enough parallelism to compute at least one run of the algorithm, we can chain these program transformations.

### 3.2 Higher-order masking

We protect our implementation against power analysis attacks by adopting the higher-order masking method of Barthe *et al.* [BDF$^+$17, Algorithm 3] at order 4 and a Trichina gate at order 2. At order 4 and following Journault and Standaert [JS17] bitsliced implementation of AES, we conservatively refresh the output of every multiplication to achieve composability through strong non-interference (as per [BDF$^+$17, Table 4]).

Our bitsliced version of the (order-2) Trichina gate has a built-in refresh at the output. Specifically, if $\mathbf{x}$ and $\mathbf{y}$ are two-share inputs and $\mathbf{r}$ is a two-share random vector, then the two-share output is obtained as

$$\mathbf{z} = \mathbf{x}.\mathbf{y} \oplus \mathbf{x}.rot(\mathbf{y}, 1) \oplus \mathbf{r} \oplus rot(\mathbf{r}, 1)$$

Optimizing this masked design by reducing the number of refresh [BBD$^+$16, BGR18] is orthogonal to the present work: our performance results serve as a pessimistic baseline; the SKIVA platform would accommodate optimized implementations – already existing or to come – just as well.

We support masking with 1, 2, and 4 shares leading to respectively unmasked, 1st-order, and 3rd-order masked implementations. By convention, we use the letter $D$ to denote the number of shares ($D \in \{1, 2, 4\}$) of a given implementation. Within a machine word, the $D$ shares encoding the $i^{\text{th}}$ bit are grouped together, as illustrated in Figure 2 for ($D \in \{1, 2, 4\}, R_s = 1$). Starting from a bitsliced design, this transformation is systematic: non-linear instructions are expanded into a masked multiplication (followed by a refresh for $D = 4$) while linear instructions are replicated over each share. The parallelism of the resulting bitsliced design is divided by $D$. The overall run-time increases with the number of

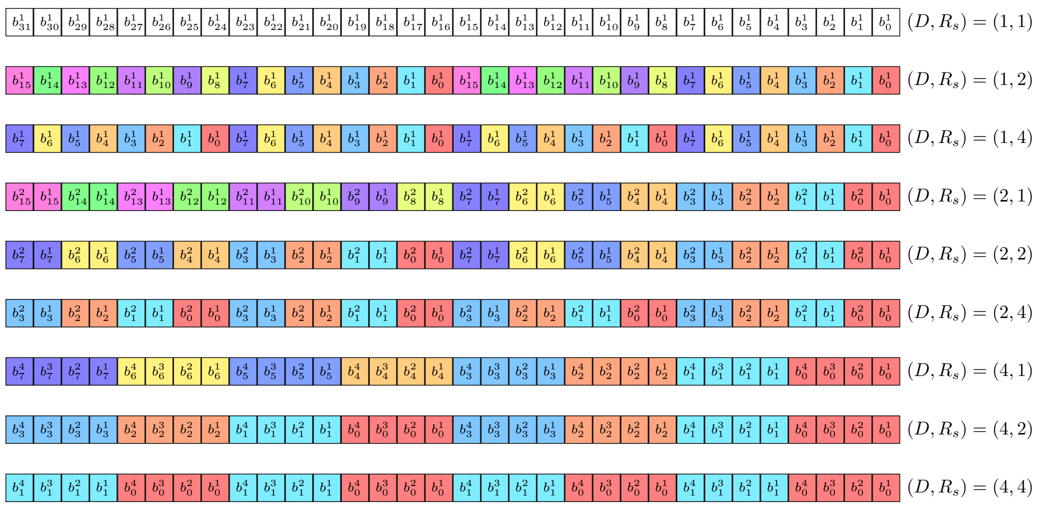


**Figure 2:** Bitslice aggregations on a 32 bit register, depending on $(D,R_s)$.

non-linear instructions and with $D$ [GJRS18]. From a security point-of-view, the bitsliced approach is a natural fit for software implementations, where leakage originates from a CPU using CMOS technology for which leakage is mostly transition-based. Understanding and controlling transition-based leakage in general is an arduous task [JS17]. However, by segregating the various shares in fixed, physically isolated slices of the registers, bitslicing provides a simpler model to reason about and control interferences across shares.

## 3.3 Intra-instruction redundancy

We protect our implementation against data faults using intra-instruction redundancy (IIR) [PYGS16, LCFS18, CSN+17]. We may duplicate a single slice into one (*i.e.* no spatial redundancy), two or four slices, checking at the end of each round that all (redundant) slices agree upon the result. By convention, we use the letter $R_s$ to denote the spatial redundancy ($R_s \in \{1, 2, 4\}$) of a given implementation. Within a machine word, the $R_s$ duplicates of the $i^{\text{th}}$ bit are interspersed every $32/R_s$ bits, as illustrated in Figure 2 for $(D = 1, R_s \in \{1, 2, 4\})$. Note that this scheme alone does not protect against control faults such as instruction skip: because redundancy is spatial and not temporal, skipping a (parallel, bitwise) operation would affect all the redundant slices simultaneously.

Starting from a bitsliced design, this transformation is systematic and exists in two forms. One can either implement a *direct* redundant implementation, in which the duplicated slices contain the same value, or a *complementary* redundant implementation, in which the duplicated slices are complemented pairwise. For example with $R_s = 4$, we can have 4 exact copies (direct redundancy) or 2 exact copies and 2 complementary copies (complementary redundancy). The direct-redundancy scheme requires no change to the code: we merely have to duplicate the inputs upon calling the protected code and testing for equality of the output slices. The complementary-redundancy scheme requires special support from the processor: a logical instruction in the original bitsliced design must be translated into an instruction that performs this operation on half of the slices and the complement operation on their redundant copies. We describe such an instruction set in Section 4. The parallelism of the resulting bitsliced design is divided by $R_s$. The overall latency is left unchanged, hence we expect the throughput of the resulting cipher to be divided by $R_s$.

In practice, we will favor complementary redundancy instead of direct redundancy. First, it is less likely for complemented bits to flip to consistent values due to a single fault injection. For instance, timing faults during state transition [ZDCT13] or memory accesses [BGV11] follow a random word corruption or a stuck-at-0 model. Second, following the wave dynamic differential logic (WDDL) approach [TV04], this enables us to expose the same Hamming weight for each individual register throughout the entire execution of the cipher, which has been shown to reduce power leakage compared to direct redundancy [BJHB18].

## 3.4 Temporal redundancy

We protect our implementation against control faults using temporal redundancy (TR) across rounds [PYGS16]. This technique consists in pipelining the execution of 2 consecutive rounds in 2 aggregated slices (Figure 3a). By convention, we use the letter $R_t$ to distinguish implementations with temporal redundancy ($R_t = 2$) from implementations without ($R_t = 1$). For $R_t = 2$, half of the slices compute round $i$ while the other half compute round $i-1$. The corresponding pseudo-code is shown in Figure 3b. The function `init_round` starts the pipeline by filling half of the slices (in `state`) with the output of the first round of AES, and the other half with the output of the initial `AddRoundKey`. At the end of round $i+1$, we have re-computed the output of round $i$ (at a later time): we can therefore compare the two results (using the `check` procedure) and detect control faults based on the different results they may have produced. If a control fault has impacted the output of round $i$ during iteration $i$ or (exclusively) $i+1$, it will necessarily be detected. To go unnoticed, the fault must be repeated in both rounds – while not impacting the subsequent round computed at iteration $i+1$ – so as to yield the same output in both iterations.

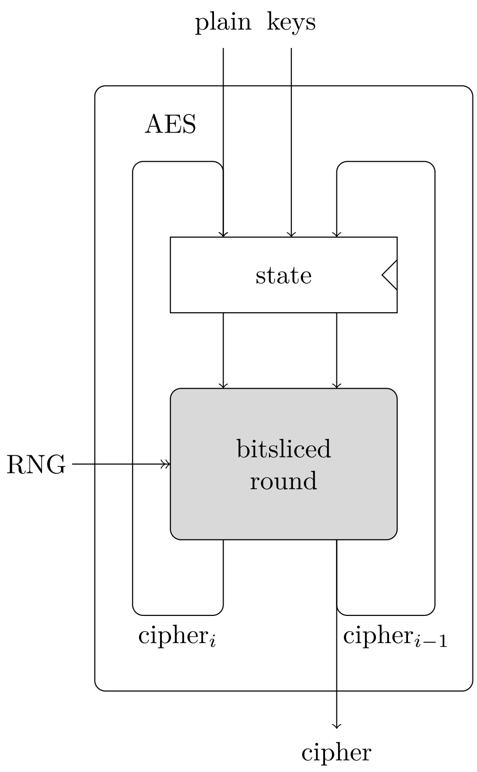
Note that unlike usual implementations of temporal redundancy, such as instruction duplication [PHBC17], this technique does not increase code size: the same instructions compute both rounds at the same time. The last round, omitted from Figure 3b, is different from the others as it does not perform `MixColumn`, and must therefore be computed twice in a non-pipelined fashion (*i.e.*, using instruction duplication), after which a final `check` is performed.

Whereas pipelining protects the inner round function, faults remain possible on the control path of the loop itself. For instance, one may attempt to sidestep the rounds by (data) faulting the loop counter or (control) faulting the conditional jump to reach the end of the loop earlier or later than desired. We protect against these threats by applying folklore loop hardening techniques: we spatially duplicate the 4-bit counter to protect against data faults and duplicate the control structure of the loop (Figure 3b).

By exploiting the iterative nature of the algorithm and the bitsliced implementation of a round, we obtain a data and control fault protection at minimal expense in code size and execution time. Since the parallelism of the inner round is divided by $R_t$, we expect the overall throughput of the cipher to be divided by $R_t$. Beside throughput, this implementation is also space-efficient (our target platform features only 128 kB of RAM): the protection against control faults piggybacks on the protection against data faults, thus avoiding instruction duplication and keeping code size in check.

## 3.5 Combining higher-order masking, IIR and TR

The protections described in the previous sections transform bitsliced designs into bitsliced designs, merely reducing parallelism (and thus throughput) in the process. As a result, they naturally compose: given a number of shares $D \in \{1, 2, 4\}$, a spatial redundancy $R_s \in \{1, 2, 4\}$ and a temporal redundancy $R_t \in \{1, 2\}$, we can systematically derive an implementation immune to power analysis and/or data faults and/or control faults, processing $32/(R_t \times R_s \times D)$ blocks at a times. The 9 possible layouts for $(D, R_s, R_t = 1)$ are illustrated in Figure 2.



(a) Schematic view

```
void AES_secure(uint plain[128], uint keys[11][128],
                uint cipher[128]) {
  uint state[128];
  // Aggregated bitslice 'state': plain and first round
  init_round(state, plain, keys[0], keys[1]);
  // Data-duplicated loop counter, increment and guard
  int round_cpt = 1 | (1 << 4);
  const int incr = 1 | (1 << 4);
  const int last_round = 9 | (9 << 4);

  // Duplicated loop structure
  while (1) {
    while (1) {
        // Retrieve key from duplicated round index
        uint[128] round_key = load_key(keys, round_cpt);
        // Compute current and previous round in parallel
        AES_round_bitsliced(state, round_key, plain);
        // Check temporal redundancy
        check(state, plain);
        memcpy_secure(plain, state, 128*sizeof(uint));
        // Increment data-duplicated counter
        round_cpt += incr;
        // Duplicated loop exit
        if (round_cpt == last_round) break;
    }
    if (round_cpt == last_round) break;
  }
  // last round twice, checked for temporal redundancy
  (..)
}
```

(b) Temporally redundant run-time

**Figure 3:** Compact and protected AES skeleton

The modularity of our approach paves the way for pay-as-you-go countermeasures: depending on the execution context and security requirements of our cipher, we can decide to adopt a more or less aggressive set of parameters $(D, R_s, R_t)$. Different protections are obtained by combination of the 3 elementary protection mechanisms available. Let us first consider the most secure implementations and justify their relevance. In a setting where multiple cycle-accurate and bit-precise faults are possible [NHH+16], we would recommend an implementation with $(D \in \{2, 4\}, R_s \in \{2, 4\}, R_t = 2)$. If faults cannot be reliably repeated in a cycle-accurate and/or bit-precise manner, then $(D \in \{1, 2, 4\}, R_s = 1, R_t = 2)$ is sufficient. A physically isolated device forbids power analysis but is not necessarily immune to faults [TSS17, KDK+14] altogether: in this setting, one could adopt $(D = 1, R_s \in \{2, 4\}, R_t = 2)$. We may further strengthen our hypothesis about the device and thus relax our security requirements. For example, we may dispense from redundancy altogether if the device is physically protected against probes [YGD+16, CBD+15, LLF16], yielding $(D \in \{2, 4\}, R_s = 1, R_t = 1)$. Or we may assume that the underlying architecture provides hardware support enforcing control-flow integrity [WWM15, dCKC+16], in which case temporal redundancy can be disposed of but not spatial redundancy, covering the cases where $(D \in \{1, 2, 4\}, R_s \in \{2, 4\}, R_t = 1)$. We have thus mapped the entire design space.

**Table 1:** Proposed ISE. These instructions are added to the standard SPARC-V instruction set, occupying unused opcodes. Symbols in the instruction format - `rs1`, `rs2`, `rd` are registers. `imm` is an immediate operand. The "Type" column shows what opcode group was used for each instruction. Appendix A lists the functional specification for each instruction.

| Semantics | Instruction format | Immediate | Type |
|---|---|---|---|
| Normal → Bitslice | `TR2 rs1, rs2, rd` | | logic |
| Bitslice → Normal | `INVTR2 rs1, rs2, rd` | | ld/st |
| Slice Rotation | `SUBROT rs, imm, rd` | $D$ | logic |
| Redundancy Generation | `RED rs, imm, rd` | $R_s$ | logic |
| Redundancy Checking | `FTCHK rs, imm, rd` | $R_s$ | logic |
| Redundant AND ($R_s$=2) | `ANDC16 rs1, rs2, rd` | | logic |
| Redundant XOR ($R_s$=2) | `XORC16 rs1, rs2, rd` | | logic |
| Redundant XNOR ($R_s$=2) | `XNORC16 rs1, rs2, rd` | | ld/st |
| Redundant AND ($R_s$=4) | `ANDC8 rs1, rs2, rd` | | logic |
| Redundant XOR ($R_s$=4) | `XORC8 rs1, rs2, rd` | | logic |
| Redundant XNOR ($R_s$=4) | `XNORC8 rs1, rs2, rd` | | ld/st |

## 4 Implementation aspects

In the previous section, we have introduced several bitslice aggregation schemes providing multiple levels of side-channel resistance and fault-attack resistance, depending on a selected number of shares, level of spatial redundancy, and level of temporal redundancy $(D, R_s, R_t)$. In this section, we present the SKIVA hardware, a custom instruction set extension (ISE) tailored to support an efficient and safe implementation of these schemes. We first lay bare the hardware and software assumptions our design is operating under (Section 4.1) and expound its semantics (Section 4.2).

### 4.1 Hardware design space

Custom instructions are commonly used as performance-enhancing mechanisms [GGP08] since a single custom instruction can replace multiple standard instructions. Custom ISE is also useful to support hardware-specific side-channel countermeasures, such as mask generation [TG07] or hiding [RCS+09]. Adding a new instruction to a processor requires modification of the processor data-path as well as modification of the processor software toolchain. With the advent of open platforms such as RISC-V and the widespread availability of programmable logic (FPGA), instruction extensions have become a practical methodology. For example, XCrypto [MPP19] defines instruction extensions for RISC-V. CRISP [GGP08] is another effort to add native support for bitslicing in a processor design. CRISP defines three new instructions, each with six operands. Their custom instruction datapath relies on two programmable lookup tables with four input bits and one output bit. However, these instructions only deal with bitslicing, and they do not offer redundancy nor support for countermeasures.

**Design.** The design of new instructions involves a trade-off between a specialized, application-specific solution and a general-purpose, universal solution. Each new custom instruction must serve as many different applications as possible. We added new instructions to SKIVA to support computing on aggregated bitslices in three different areas. First, they help with the conversion from normal representation to bitsliced form and back. Second, they handle subword-operations for the computation of non-linear operations on two or four shares ($D \in \{2, 4\}$). Third, they handle subword-operations for spatially redundant computations and fault checking ($R_s \in \{2, 4\}$). The new instructions

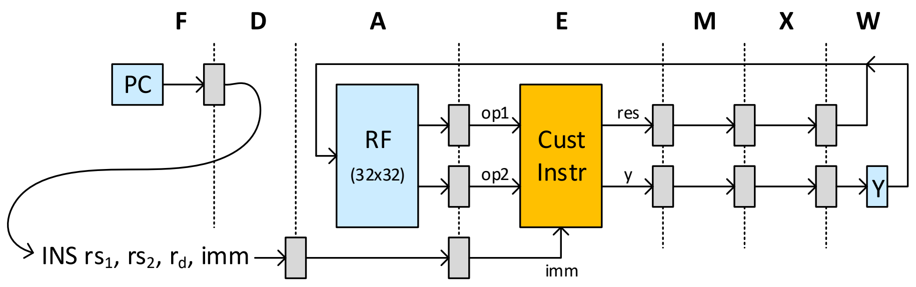


**Figure 4:** Integrated in the regular 7-stage pipeline as a new execution stage.

are summarized in Table 1 and will be described in detail in further subsections. Appendix A provides their functional specification. These new instructions are orthogonal; they can be used in a mix-and-match fashion to obtain the desired level of sharing and redundancy. We integrated the new instructions on the SPARC-V instruction set of the open-source LEON3 processor and software toolchain [Res18].

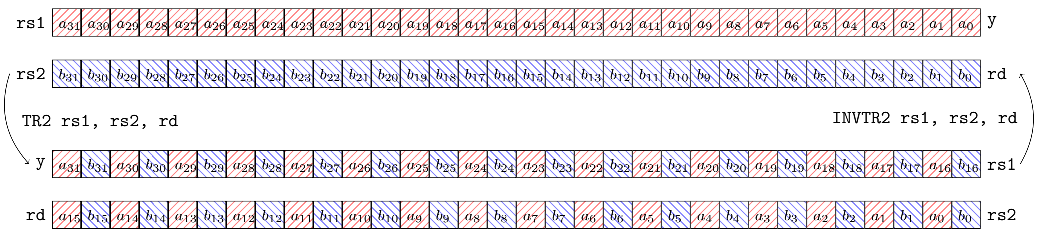
**Hardware integration.** Figure 4 illustrates the integration of the custom datapath into the seven-stage RISC pipeline. The instructions follow a two-inputs, two-outputs operand format, encoded as two source registers, a destination register and an immediate field (`INS rs1, rs2, rd, imm`). The upper 32-bit output of the custom instruction is transferred to the `Y`-register, a register which is used for SPARC-V instructions with 64-bit output such as the regular data multiplication. The integration of custom-hardware deep into the pipeline necessitates the use of simple and fast datapath hardware. However, these instructions benefit from the same performance advantages as regular instructions including a typical throughput of 1 instruction per cycle and minimal stall effect thanks to forwarding [PH12].

The new instructions are mapped into unused opcodes of the SPARC-V instruction set [SI92]. This standard instruction set recognizes three different formats. The newly added instructions belong to the third format, sharing the same opcode space as the instructions for load/store, logic, and arithmetic, among others. Because all of the proposed instructions correspond to simple logic manipulations, we integrated them directly into the existing logic/shift group and the load/store group of SPARC. Within the logic/shift group, we identified eight unused opcode locations, and we allocated the most frequently needed instructions in these unused spaces. The remaining three instructions were moved into the load/store group. The last column of Table 1 identifies the allocation for each new instruction. Since we did not replace any existing SPARC instruction, SKIVA is backward binary-compatible with existing LEON applications. The new instructions add minimal overhead to the design. In terms of 180nm standard cell ASIC technology, we added 1250 gate-equivalent to the design, which amounts to 3% of the area of the integer unit of SKIVA.

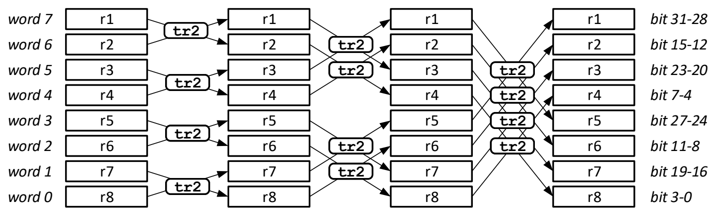
**Software integration.** We integrated the new instructions into the software toolchain of SKIVA by extending the assembler. The new mnemonics were then integrated into the application in C through inline assembly coding. Because the custom-instruction format is compatible with that of existing, standard SPARC-V instructions, they benefit from off-the-shelf compiler optimizations.

## 4.2 Hardware support for aggregated bitslice operations

In the following, we describe each group of instructions, with emphasis on their semantics. The formal definition of each instruction is given in Appendix A.



**(a)** Semantics of TR2 and INVTR2.



**(b)** Example of an 8-bit bitslice transposition using 8 registers.

**Figure 5:** Transposition and its inverse

**Instructions for bitslicing.** We introduce two instructions to transpose data into their bitsliced representation (Figure 5a). The first instruction, `TR2 rs1, rs2, rd`, performs an interleaving of the bits of two source registers into two output registers. This interleaving can be thought of as a 2-bit transposition, as it places bits within the same column of register `rs1` and `rs2` in adjacent positions of the output registers `rd` and `y`. The second instruction, `INVTR2 rs1, rs2, rd`, performs the inverse operation. Bitslice transposition for an arbitrary number of bits is achieved through repeated application of `TR2`. For example, Figure 5b shows an 8-bit transposition, that is, a bitslice conversion of 8-bit subwords within the input registers `r1` to `r8`. Twelve applications of `TR2` in a butterfly-like diagram yield the desired result. In general, for a $2^n$-bit transition, $n.2^{n-1}$ applications of `TR2` are needed. The reader may notice that the bitslice ordering of the output exhibits the same shuffling effect as for a Fast-Fourier Transform. This effect is dealt with through proper register ordering before transposition.

To create aggregated bitslices ($R_s > 1$ or $D > 1$), we pre-process the source registers (in non-bitsliced form) by duplicating them first and then tranposing them to bitsliced form. The side-channel protection and fault-detection of SKIVA is not active during bitslice conversion but we check their consistency after transposition and before encryption.

**Instructions for higher-order masking.** To combat side-channel leakage, SKIVA supports two-share and four-share implementations of bitsliced algorithms, which provides first-order and third-order masked side-channel resistance. The shares are located in adjacent bits of a processor register. We use Boolean masking, so that the XOR of all shares yields the unmasked value. Linear operations on an ensemble of shares are computed as the linear operation on each individual share. Linear operations are done using bitwise operations on the two-share and four-share representation. As presented in Section 3.2, we implement the secure multiplication (AND) using the design of Barthe *et al.* [BDF+17] for third-order masking and the design of Trichina [Tri03] for first-order masking. The secure OR operation is implemented as the De Morgan's equivalent of a secure AND.

Computing a secure multiplication over multiple shares requires the computation of the partial share-products. For example, the secure multiplication of the two-share slices $(a_1, a_0)$ with the two-share slices $(b_1, b_0)$ requires the partial products $a_1.b_1$, $a_1.b_0$,

$a_0.b_1$, and $a_0.b_0$. To align the slices for the cross-products, we implement a slice rotation instruction `SUBROT rs, imm, rd`. This instruction transforms the two-share slices $(a_1, a_0)$ into $(a_0, a_1)$. The same instruction `SUBROT` can also handle a four-share design, which transforms $(a_3, a_2, a_1, a_0)$ into $(a_2, a_1, a_0, a_3)$. The details of this instruction are given in listing A.3 in Appendix A.

The programming of side-channel protected bitsliced code using `SUBROT` assumes the following specific programming rules. Attention has to be paid to side-effects of shared storage elements in the architecture. Balasch *et al.* [BGG+14] have shown that a $d$-th order security proof against value-based leakage leads to a $\lfloor \frac{d}{2} \rfloor$-th proof against transition-based leakage. Papagiannopoulos *et al.* [PV17] identified three practical cases in micro-controller code, where such transition-based leakage occurs. The most obvious source of transition-based leakage is the overwriting of registers, since the power dissipation of overwriting the register is proportional to the Hamming distance between the former and the new value. They also observe transition-based leakage by overwriting of shared memory locations. Finally, they observe a "neighbour leaking" effect where operations on one register cause leakage from another.

In a bitslice design, different shares are stored in different bits. Transition-based leakage will occur when one bitslice overwrites another, and this can unmask the shares as follows. Assume a two-share bitslice design $(a_0, a_1) = (r \oplus v, r)$, with $r$ a random bit and $v$ a secret bit. Then writing the value $(a_1, a_0)$ into a register holding $(a_0, a_1)$ leads to unmasking. In this case, the Hamming distance is $(a_0 \oplus a_1, a_1 \oplus a_0) = (r \oplus v \oplus r, r \oplus v \oplus r) = (v, v)$. This example directly applies to `SUBROT`, when this instruction would write its output into its own source register.

To avoid these known sources of transition-based leakage, and to minimize the risk of (undesired) unmasking resulting from this leakage, we applied the following conservative strategy. (1) For $D = 4$, we refresh the masks at the output of every secure multiplication (AND) using Barthe's parallel refreshing algorithm [BDF+17]. For $D = 2$, the refresh is implicit due to the construction of the Trichina gate; (2) We avoid reusing registers within the secure multiplication by constraining the set of registers that the compiler is allowed to use. This ensures that `SUBROT` will never overwrite its own input. In addition, after the result of a `SUBROT` instruction is used, we clear that register to prevent later overwriting by another instruction. Figure 6 shows an example of a first-order and a third-order secure multiplication. (3) We maintain strict separation between registers used for the masked algorithm (*i.e.* AES), and registers used for mask generation and mask distribution. This ensures that registers containing masked data cannot be overwritten by registers directly related to random masks. A separation for transition-base leakage between two registers `ra` and `rb` means that neither register is allowed to overwrite the other one.

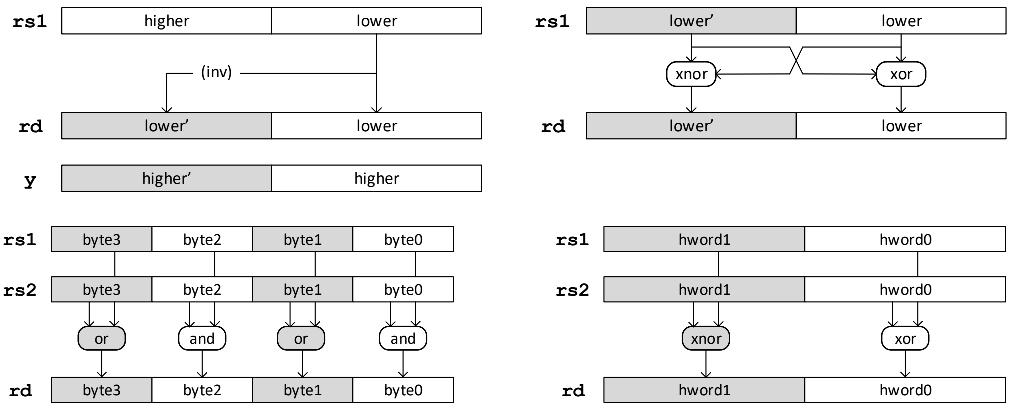


**Figure 7:** (a) Example of RED on half-word (top, left). (b) Example of FTCHK on half-word (top, right). (c) Example of ANDC8 (bottom, left). (d) Example of XORC16 (bottom, right).

```
# input: %i2 (a), %i3 (b), %i4 (random)
# output: %o0
AND    %i3, %i2, %o5  # partial product 1
SUBROT %i2,   2, %l0  # share-rotate
AND    %i3, %l0, %o3  # partial product 2
XOR    %l0, %l0, %l0  # clear SUBROT output
XOR    %o5, %i4, %o2  # random + parprod 1
XOR    %o2, %o3, %o1  #        + parprod 2
SUBROT %i4,   2, %l1  # parallel refresh
XOR    %o1, %l1, %o0  #          output
```

**(a)** First-order secure multiplication

```
# input: %l7 (a), %g1 (b), %g4 (random), %g2 (random)
# output: %i1
AND    %l7, %g1, %i3 # partial product 1
SUBROT %l7, 4, %o1   # share-rotate
AND    %g1, %o1, %i2 # partial product 2
SUBROT %g1, 4, %o0   # share-rotate
AND    %o0, %l7, %i0 # partial product 3
SUBROT %o1, 4, %l0   # share-rotate
AND    %g1, %l0, %o7 # partial product 4
XOR    %o1, %o1, %o1 # clear SUBROT output
XOR    %o0, %o0, %o0 # clear SUBROT output
XOR    %l0, %l0, %l0 # clear SUBROT output
XOR    %i3, %g4, %o5 # random + parprod 1
XOR    %o5, %i2, %o4 #        + parprod 2
XOR    %o4, %i0, %o3 #        + parprod 3
SUBROT %g4, 4, %l1   #
XOR    %o3, %l1, %o2 #        + rot(random)
XOR    %o2, %o7, %g3 #        + parprod 4
XOR    %g2, %g3, %i5 # output refresh
SUBROT %g2, 4, %l2   #
XOR    %l2, %i5, %i1 #
```

**(b)** Third-order secure multiplication

**Figure 6:** Secure multiplication using SUBROT

**Instructions for fault redundancy checking.** We present the instructions related to fault redundancy in two groups. The first is related to generation and checking of fault-redundant slices, while the second is related to computations. The redundant bits with respect to fault injection are stored in adjacent bytes of half-word. Figure 7(a) shows the example of a halfword operation to generate redundant data, while Figure 7(b) shows the example of a halfword operation to verify redundant data.

The `RED rs1, imm, rd` instruction generates redundant data. The redundant copy is stored in the upper halfword ($R_s = 2$) or in the three upper bytes ($R_s = 4$). The redundant portion can be either a direct or else a complement of the original data. There are six variants of `RED rs1, imm, rd`. Two of them support dual redundancy ($R_s = 2$), they duplicate the lower and upper halfword, in direct or complementary form. Four additional variants support quadruple redundancy ($R_s = 4$), and they quadruple the lower two bytes or the upper two bytes, each in direct or complementary form. Listing A.4 gives a formal definition of these instructions.

The `FTCHK rs1, imm, rd` instruction verifies the consistency of the redundant data. This instruction generates a fault-flag in redundant form (over $R_s$ bits, Appendix A.11), which can be used to drive a fault condition test. Figure 7(b) illustrates the case of a dual-redundancy check on direct data. The fault-check is evaluated in a redundant manner, so that the fault-check itself can detect fault injection on its own check. The expected faultless result of the instruction example in Figure 7(b) is `0xFFFF0000`. There are four variants of this instruction, for either dual ($R_s = 2$) or quadruple redundancy ($R_s = 4$), and direct or complementary redundancy.

**Instructions for fault-redundant computations.** Computations on direct-redundant bitslices can be done using standard bitwise operations. However, for complementary-redundant bitslices, the bitwise operations have to be adjusted to complement-operations. The complement-redundant data format can be introduced at the halfword boundary ($R_s = 2$) or at the byte boundary ($R_s = 4$). We opted to provide support for bitwise AND, XOR and XNOR on these complement-redundant data formats. Figure 7(c-d) illustrates the case of `ANDC8` and `XORC16`. Their detailed behavior is given in Appendix A.

## 5 Results

This section evaluates the performance and side-channel security of AES on SKIVA. Next, we analyze the fault coverage of applications on SKIVA under the assumed fault model.

### 5.1 Performance evaluation

Our experimental evaluation has been carried on a prototype of SKIVA deployed on the main FPGA (Cyclone IV EP4CE115) of an Altera DE2-115 board. The processor is clocked at 50 MHz and has access to 128 kB of RAM. Our performance results are obtained by running the desired programs on bare metal. We assume that we have access to a TRNG that frequently fills a register with a fresh 32-bit random string. We use a software pseudo-random number generator (32-bit xorshift) to emulate a TRNG refreshed at a rate of our choosing. We checked that our experiments did not overflow the period of the RNG.

Several implementations of AES are available on our 32-bit, SPARC-derivative processor, with varying degrees of performance. A straightforward byte-oriented implementation (supplementary material) takes 77 C/B whereas an optimized 32-bit T-box implementation (supplementary material) takes 23 C/B. Both implementations are prone to timing attacks. The constant-time, byte-sliced implementation (using only 8 variables to represent 128 bits of data) of BearSSL [Por] performs at 48 C/B. Our bitsliced implementation (using 128 variables to represent 128 bits of data) (supplementary material) performs favorably at 44 C/B while weighing 7772B: despite a significant register pressure (128 live variables for 32 machine registers), the rotations of `MixColumn` and the `ShiftRows` operations are compiled away. This bitsliced implementation serves as our baseline in the following.

**Micro-benchmarks.** The instructions `TR2`, `INVTR2`, `RED` and `SUBROT` were introduced solely for performance reasons. We evaluate their associated performance benefits by micro-benchmarking them against an equivalent, purely software emulation. The instructions `TR2`/`INVTR2` improve performance by $\times 3.64$ whereas `SUBROT` improves performance by $\times 4.2$. The instruction `RED` improves performance from $\times 2.7$ for $R_s = 2$ to $\times 14.1$ for $R_s = 4$. These results are consistent with the number of instructions necessary to emulate each instruction (Appendix B). The impact of memory transfers (which takes a significant portion of the computation time, independently of the instruction set) somewhat reduces the absolute benefits of `TR2`/`INVTR2` instructions: a full, bitsliced transposition takes 426 cycles with software emulation while it takes 302 cycles with custom instructions, yielding a speedup of $\times 1.4$ with custom instructions.

**Code size (AES).** We measure the impact of our hardware and software design on code size, using our bitsliced implementation of AES (Section 3) as a baseline. Our hardware design provides us with native support for spatial, complementary redundancy (`ANDC`, `XORC`, and `XNORC`). Performing these operations through software emulation would result in a $\times 1.2$ (for $D = 2$) to $\times 1.3$ (for $D = 4$) increase in code size. One must nonetheless bear in mind that the security provided by emulation is *not* equivalent to the one provided by native support. The temporal redundancy ($R_t = 2$) mechanism comes at the expense of a small increase (less than $\times 1.06$) in code size, due to the loop hardening protections as well as the checks validating results across successive rounds. The higher-order masking comes at a reasonable expense in code size: going from 1 to 2 shares increases code size by $\times 1.4$ whereas going from 1 to 4 shares corresponds to a $\times 1.8$ increase. A fully protected implementation ($D = 4, R_s = 4, R_t = 2$) thus weighs 14048 bytes.

**Throughput (AES).** We report on the impact of our hardware and software design on the performance of our bitsliced implementation of AES (Section 3). To do so, we evaluate the performance of our 18 variants of AES, for each value of ($D \in \{1, 2, 4\}, R_s \in \{1, 2, 4\}, R_t \in \{1, 2\}$). To remove the influence of the TRNG's throughput from the performance evaluation, we assume that its refill frequency is strictly higher than the rate at which our implementation consumes random bits. In practice, a refill rate of 10 cycles for 32 bits is enough to meet this requirement.

We report our performance results[1] in Table 2. As expected, for $D$ and $R_t$ fixed, the throughput decreases linearly with $R_s$. Comparing Table 2a with Table 2b at fixed $R_s$, we notice that the throughput decreases by a factor $\times 2.5$ ($D = 4$) to $\times 3$ ($D = 1$): temporal redundancy mechanically divides the throughput by a factor 2, on top of which one must account for the overhead of scheduling and checking the redundant slices. Note that this overhead is less acute as $D$ increases since more time is spent computing each AES round (and, thus, relatively less time is spent in the runtime implementing temporal redundancy). At fixed $D$, we also notice that the variant ($D, R_s = 1, R_t = 2$) (temporal redundancy by a factor 2) exhibits similar performances as ($D, R_s = 2, R_t = 1$) (spatial redundancy by a factor 2). However and crucially, both implementation are *not* equivalent from a security standpoint: as discussed in Section 5.3, the former offers weaker security guarantees than the latter. Similarly, at fixed $D$ and $R_s$, we may be tempted to run twice the implementation ($D, R_s, R_t = 1$) rather than running once the implementation ($D, R_s, R_t = 2$): once again, the security of the former is reduced compared to the latter since temporal redundancy ($R_t = 2$) couples the computation of 2 rounds within each instruction, whereas pure instruction redundancy ($R_t = 1$) does not. At fixed $R_s$ and $R_t$, going from $D = 1$ to $D = 2$ implies a serious performance toll: first, the throughput is mechanically divided by a factor 2; second, non-linear instructions must be expanded into secure ones; third, there is a significant run-time overhead induced by masking, such as creating shares, fetching random numbers, *etc.* Comparatively, going from 2 to 4 shares, is less expensive since these run-time overheads are identical.

In Figure 2c and Figure 2d, we report the speedup offered by the custom instruction set compared to a software emulation of these instructions. For large values of $R_s$ or $D$, the custom instruction set yields a speedup between $\times 1.4$ to $\times 1.6$, which is a reasonable expectation for a fine-grain custom-instruction based hardware acceleration mechanism [IL07]. On the one hand, custom instructions can be emulated in 2 instructions on

[1] To fully account for the 3 dimensions of our design space ($D$, $R_s$, and $R_t$), we present our results in a tabular form – rather than graphical – to avoid biasing the interpretation toward 2 particular dimensions, at the exclusion of the third one.

**Table 2:** Exhaustive evaluation of the AES design space

| $R_t = 1$ | | $D$ = 1 | $D$ = 2 | $D$ = 4 |
|---|---|---|---|---|
| $R_s$ | 1 | 44 C/B | 183 C/B | 621 C/B |
| | 2 | 89 C/B | 447 C/B | 1615 C/B |
| | 4 | 169 C/B | 847 C/B | 3042 C/B |

**(a)** Throughput ($R_t = 1$)

| $R_t = 2$ | | $D$ = 1 | $D$ = 2 | $D$ = 4 |
|---|---|---|---|---|
| $R_s$ | 1 | 127 C/B | 470 C/B | 1507 C/B |
| | 2 | 262 C/B | 1122 C/B | 3838 C/B |
| | 4 | 513 C/B | 2148 C/B | 7272 C/B |

**(b)** Throughput ($R_t = 2$)

| $R_t = 1$ | | $D$ = 1 | $D$ = 2 | $D$ = 4 |
|---|---|---|---|---|
| $R_s$ | 1 | ×1.07 | ×1.51 | ×1.54 |
| | 2 | ×1.41 | ×1.51 | ×1.57 |
| | 4 | ×1.50 | ×1.51 | ×1.59 |

**(c)** Speedup w/ custom instructions ($R_t = 1$)

| $R_t = 2$ | | $D$ = 1 | $D$ = 2 | $D$ = 4 |
|---|---|---|---|---|
| $R_s$ | 1 | ×1.06 | ×1.39 | ×1.40 |
| | 2 | ×1.35 | ×1.45 | ×1.50 |
| | 4 | ×1.46 | ×1.48 | ×1.55 |

**(d)** Speedup w/ custom instructions ($R_t = 2$)

average: at best, our speedup is at most ×2. On the other hand, relatively few custom instructions are used: they appear mostly in the S-box, whereas the remainder of the cipher consists of linear operations and memory transfers. This is consistent with previous custom cryptographical ISE, such as CRISP [GGP08] where a speedup of ×1.36 was reported. Note, once again, that both implementations are *not* comparable from a security standpoint: the security argument of the former is simpler than the latter while a successful fault against the former requires a more powerful adversary than the latter.

## 5.2 Side-channel analysis

To show the security of our masking scheme, we test SKIVA on the main FPGA of SAKURA-G board running at 9.8MHz and powered at 5V by an external power generator. We use a LeCroy WaveRunner 610Zi oscilloscope, sampling 250M samples/sec. To limit the noise level, we use a low-pass filter with a cutoff frequency of 81MHz on the power probe. Furthermore, to have more accurate power traces, we set the scope to average five traces and execute each encryption five times on SKIVA. To trigger the scope, we assign one GPIO pin of LEON to a header pin on SAKURA-G board. We program a C implementation of AES on SKIVA. This C code gets a plaintext from a PC through UART and runs the encryption on the plaintext five times. The AES code sets and resets the trigger before and after the encryption steps described in the following subsections. The ciphertext is sent back to the PC for checking and validation of the power trace.

**Correlation power analysis.** To evaluate our design, we conduct $1^{st}$ order correlation power analysis (CPA) [BCO04] on power consumption traces of the `SubBytes` stage of the first round of AES. We use hamming weight of the `SubBytes` output as the power model. To speed up our attack, we use a sampling rate of 50M samples/sec. In this testcase, we attack a single bitslice out of 32 parallel bitslices; the unused bitslices perform constant encryption of an all-zero plaintext with an all-zero key. Our CPA attack analyzes 50K traces and confirms that $1^{st}$ order CPA on the unmasked scheme can reveal half of the key with 12K traces while it reveals all the secret key bytes with 24K traces (see Appendix C.1 for specifics). When masking is enabled, no key byte is revealed under any configuration at the maximum number of traces we considered (50K).

**Test vector leakage assessment.** To show that our $1^{st}$ and $3^{rd}$ order masked schemes are immune to power-based attacks of orders up to their masking orders, we use the TVLA methodology [GJJR11, BCD+13] and conduct the $1^{st}$ and $2^{nd}$ order t-tests on our $1^{st}$

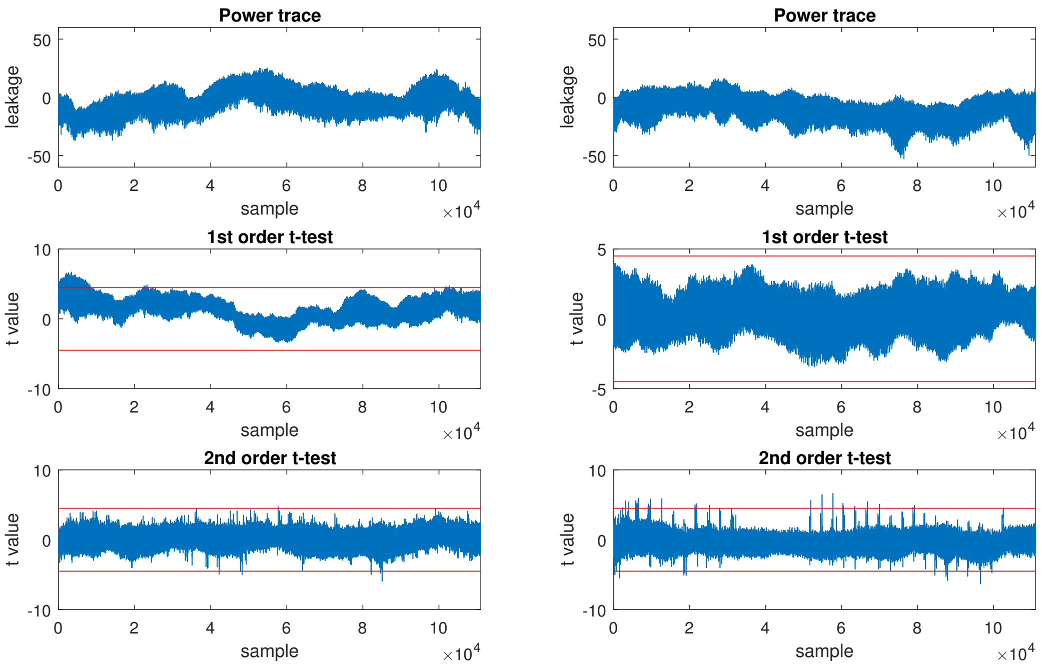


**Figure 8:** Example power trace and 1$^{st}$ and 2$^{nd}$ order t-tests of 1$^{st}$ order masked implementation. Left column: 40K fixed *vs.* 40K random traces with PRNG off. Right column: 500K fixed *vs.* 500K random traces with PRNG on.

order masked implementation and the 1$^{st}$ to 4$^{th}$ order t-tests on our 3$^{rd}$ order masked encryption. We set the trigger on one S-box in the fourth round of AES based on the observation that the t-test shows more accurate results for the second third part of AES [BCD$^+$13].

For our experiments, following our attacker model discussed in Section 2.1, we conduct the univariate non-specific fixed-*vs.*-random t-test in which a set of random inputs and a set of fixed inputs are interspersed in a random order and sent to the device. The fixed plaintext is selected such that the output of the `SubBytes` stage in the 4$^{th}$ round of AES is zero. Furthermore, for higher order t-tests, we post-process the traces [SM15b] to calculate the t-scores of the target order. We adopt the histogram methodology [RGV17] to speed up our t-test calculations. Using a threshold value of 4.5 gives us a confidence of 99.999% to test the null hypothesis that the two sets are from the same population, *i.e.* the device is not leaking information correlated to the secret data.

Figure 8 and Figure 9 show the results of the t-test on our masked implementations. The right column in Figure 8 (resp. Figure 9) indicates that our first (resp. third) order masked scheme shows no leakage of first (resp. first, second, or third) order on 500K fixed *vs.* 500K random traces while showing second (resp. fourth) order leakage as expected. The left columns show how turning the PRNG off causes the implementations to have leakage of all orders.

We conclude that, by applying the conservative approach mentioned in Section 4.2, our implementation gives $d^{th}$ order security on a $d^{th}$ order masking scheme.

**Power leakage of direct and complementary redundancy.** To compare the effect of the direct and complementary redundancy schemes on side-channel leakage, we run the following test. We make two different versions of our AES C code: (1) 16 parallel aggregated bitslices of the direct $(D = 2, R_s = 1, R_t = 1)$ scheme as the input to the first S-box in

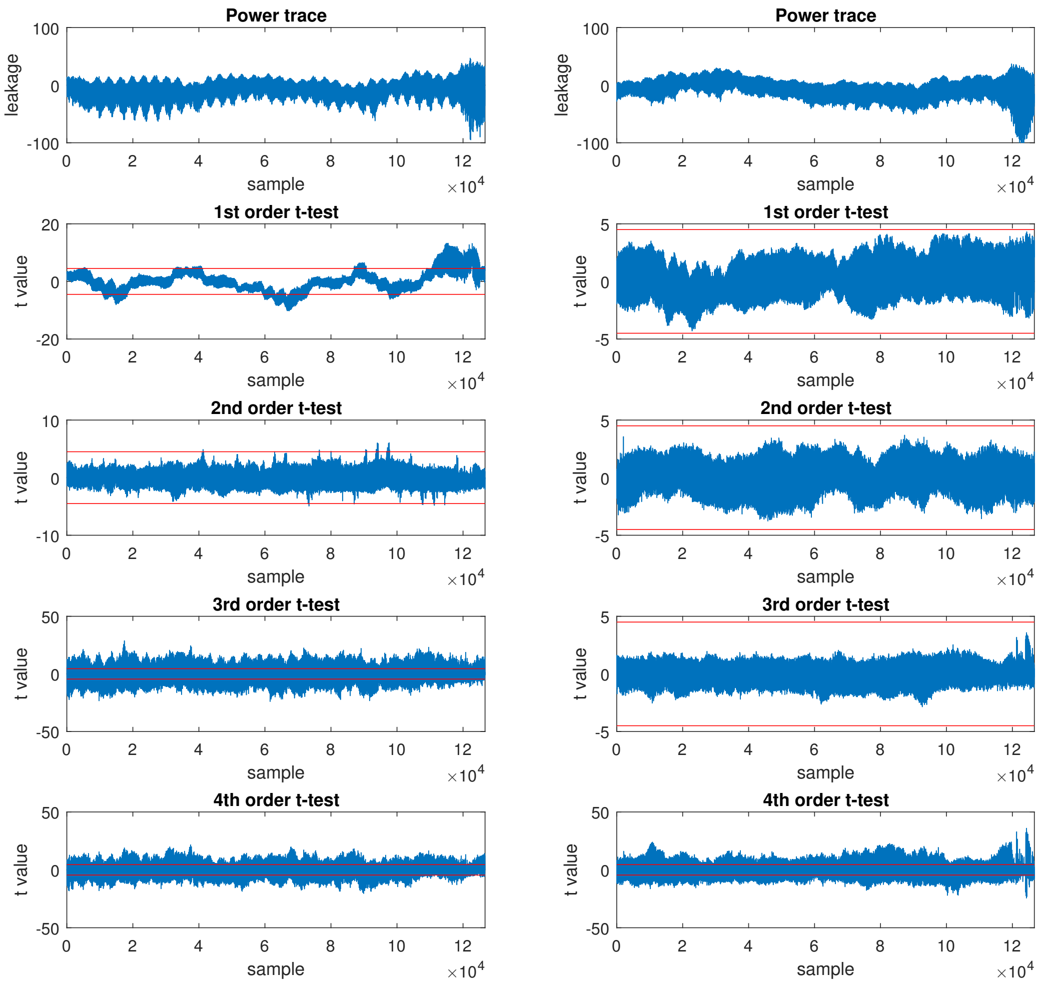
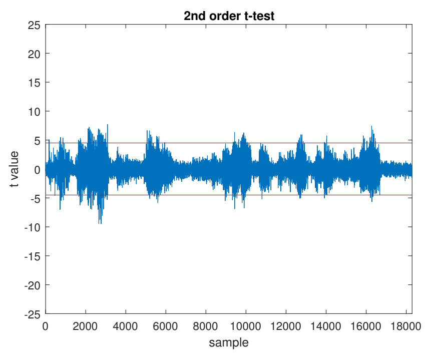
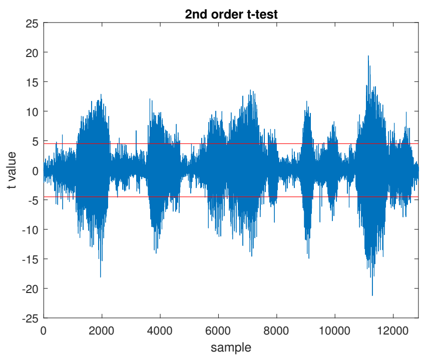


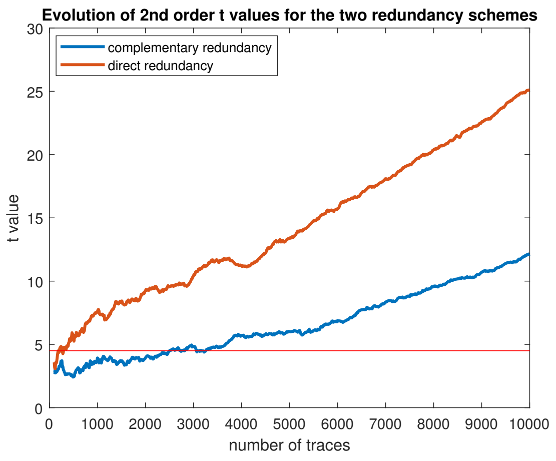
**Figure 9:** Example power trace and 1$^{st}$ to 4$^{th}$ order t-tests of 3$^{rd}$ order masked implementation. Left column: 35K fixed *vs.* 35K random traces with PRNG off. Right column: 500K fixed *vs.* 500K random traces with PRNG on.



**(a)** $1^{st}$ order masked / $2^{nd}$ order t-test Complementary redundancy (5K fixed *vs.* 5K random)



**(b)** $1^{st}$ order masked / $2^{nd}$ order t-test Direct redundancy (5K fixed *vs.* 5K random)



**(c)** Evolution of t values for $2^{nd}$ order t-test on $1^{st}$ order masked implementation with direct and complementary redundancy.

**Figure 10:** Effect of different redundancy schemes on power leakage.

the fourth round of AES; and (2) 8 parallel aggregated bitslices of the complementary ($D = 2, R_s = 2, R_t = 1$) scheme as the input to the first S-box in the fourth round of AES.

We then measure 5K traces for fixed input and 5K traces for random input and apply a second order t-test on the measured traces. To speed up our measurements, the traces were collected at 50MS/s. As expected, Figures 10a and 10b show second order leakage for both schemes. However, the direct redundancy results in much higher t-values indicating a higher probability of leakage than complementary redundancy. To make this point more clear, Figure 10c shows the evolution of t-values for the $2^{nd}$ order t-test with respect to the number of traces for both redundancy schemes. We observe that the direct redundancy shows leakage with as few as about 200 traces while the complementary redundancy shows leakage only after around 2500 traces. From this experiment, we conclude that having the complementary redundancy is better than its direct counterpart in hiding secret data from the power leakage. Note that despite exhibiting different power leakage profiles, we have confirmed that a first order t-test on both implementations shows no leakage for a non-specific test of 25K fixed *vs.* 25K random traces (Figure 11 in Appendix C.2).

## 5.3 Security analysis of data faults

In the following, we analyze the fault sensitivity of our protected implementations according to the attacker models defined in Section 2.1. Our data protection scheme relies on spatial redundancy ($R_s \in \{2, 4\}$). Faults that cannot be detected are those that affect redundant copies within a single register *in a consistent manner*, which implies either identical values in case of direct redundancy or negated values in case of complemented redundancy.

Note that this analysis is independent of whether sharing ($D$) is used or not. From the standpoint of redundancy, each share is independently protected: for example, if two shares of the same data are subjected to a bit flip, our redundancy mechanism will report an error, even though the underlying data remains unchanged ($x_1 \oplus x_2 = \overline{x_1} \oplus \overline{x_2}$).

There are different ways to achieve undetected faults, *i.e.* generate a consistent value: one may skip an instruction whose destination register already holds a consistent value; one may replace an instruction with another (*e.g.*, substitute an `ANDC` by an `XORC`); or directly perform a data fault.

If $P$ is the probability for a data fault to result in a consistent value, then the detection rate is $1 - P$. Such a probability depends on the injection technique, its parameters, the target architecture as well as physical properties of the device. In the following, we develop a theoretical analysis based on the assumption that data faults follow a stuck-at 0 or stuck-at 1 model, or uniformly distributed random byte, half-word, and word model. We then complement this analysis by an empirical evaluation of the impact of instruction skip.

**Theoretical analysis of spatial redundancy** In this analysis, we use the fault coverage ($FC$) metric [GMK12] $FC = 1 - F_{\text{undetected}}/F_{\text{total}}$ where $F_{\text{total}}$ is the total number of faults covered by the fault model and $F_{\text{undetected}}$ is the number of faults that affect the execution while escaping detection by the countermeasure.

By construction, data fault effects such as single bit set, single reset, single bit flip, byte or half-word zeroing, faulty random byte or faulty random half-word are all detected ($FC = 100\%$). Word zeroing or stuck-at 1 on complementary redundant data are also all detected ($FC = 100\%$) but direct redundancy will never detect it ($FC = 0\%$).

If the attacker injects random data faults following a uniform distribution, it means that there are $F_{\text{total}} = 2^{32}$ fault injection possibilities. For $R_s = 2$ and independently of the redundancy (direct or complementary), $2^{16}$ of those values are consistent, including the expected output. Hence $F_{\text{undetected}} = 2^{16} - 1$ and $FC = 99.99\%$. For $R_s = 4$, there are $F_{\text{undetected}} = 2^8 - 1$ faults that are left undetected, thus $FC = 99.99\%$.

For illustrative purposes, we now consider a slightly stronger attacker who may flip $p$ randomly chosen data bits. In practice, such an analysis ought to be tailored to account for the specific distribution of faults of a given injection technique on a given platform. Under this attacker model, there are $F_{\text{total}} = \binom{32}{p}$ fault injection possibilities leading to a $p$-bit flip (with $p$ an even number). For $R_s = 2$, there are $F_{\text{undetected}} = \binom{16}{\frac{p}{2}}$ faults corresponding to a $p$-bit flip that are left undetected. The lower-bound for $FC$ is reached for $p = 2$ and $p = 30$, where $FC = 96.77\%$. For $R_s = 4$, there are $F_{\text{undetected}} = \binom{8}{\frac{p}{4}}$ faults corresponding to a $p$-bit flip that are left undetected. The lower-bound for $FC$ is reached for $p = 4$ and $p = 28$, where $FC = 99.97\%$. A $p$-bit set or reset fault model leads to a 100% detection rate if complementary redundancy is used. If direct redundancy is used, then this amounts to the $p$-bit flip model. Either way the detection rate is very high.

**Experimental evaluation of temporal redundancy.** We have simulated the impact of faults on our implementation of AES. We focus our attention exclusively on control faults (instruction skips) since our above analytical model already predicts the outcome of data faults. To this end, we use a fault injection simulator based on `gdb` running through the JTAG interface of the FPGA board. We execute our implementation up to a chosen

**Table 3:** Experimental results of simulated instruction skips

| | With impact | | Without impact | | | |
|---|---|---|---|---|---|---|
| | Detected (1) | Not detected (2) | Detected (3) | Not detected (4) | Crash (5) | # of faults |
| $R_t = 1$ | 0.19% | 94.40% | 0% | 2.56% | 2.84% | 8507 |
| $R_t = 2$ | 80.75% | 0% | 7.74% | 7.96% | 3.55% | 19552 |

breakpoint, after which we instruct the processor to jump to a given address, hence simulating the effect of an instruction skip. In particular, we have exhaustively targeted every instruction of the first and last round as well as the `AES_secure` routine (for $R_t = 2$) and its counterpart for $R_t = 1$. Since rounds 2 to 9 use the same code as the first round, the absence of vulnerabilities against instruction skips within the latter means that the former are secure against instruction skip as well. This exposes a total of 1222 injection points for $R_t = 2$ and 1097 injection points for $R_t = 1$. For each such injection point, we perform an instruction skip from 512 random combinations of key and plaintext for $R_t = 2$ and 256 random combinations for $R_t = 1$.

The results are summarized in Table 3. Injecting a fault had one of five effect. A fault may yield an incorrect ciphertext with (1) or without (2) being detected. A fault may yield a correct ciphertext, with (3) or without (4) being detected. Finally, a fault may cause the program or the board to crash (5). According to our attacker model, only outcome (2) witnesses a vulnerability. In every other outcome, the fault either does not produce a faulty ciphertext, or is detected within 2 rounds. For $R_t = 2$, we verify that every instruction skip was either detected (outcome 1 or 3) or had no effect on the output of the corresponding round (outcome 4) or lead to a crash (outcome 5). Comparatively, with $R_t = 1$, nearly 95% of the instruction skips lead to an undetected fault impacting the ciphertext. In 0.19% of the cases, the fault actually impacts the fault-detection mechanism itself, thus triggering a false positive.

## 5.4 Discussion

SKIVA sets out to provide a platform for implementing cryptographic primitives resilient to combined attacks. In this section, we have evaluated a set of candidate designs for AES in terms of performance (Section 5.1) as well as security. We have carried a theoretical and empirical evaluation of the impact of faults (Section 5.3) on our designs, hence quantifying their adequacy with respect to our "fault attacker model" (Section 2.1). We have also carried out an empirical evaluation of the security of our masking scheme through CPA and the TVLA methodology, hence quantifying its adequacy with respect to our "side-channel attacker model" (Section 2.1). Besides, we have quantified the amplification of side-channel leakage induced by the fault protection mechanism, hence validating our "combined attacker model" (Section 2.1). Admittedly, our combined attacker model exposes a narrow attack surface, excluding an attacker actively mitigating the SCA countermeasures or drawing conclusions from the distribution of masked values (SIFA). As the design and implementation of protections against such attacks mature, we will be able to integrate them in a (software) implementation of AES, leaving SKIVA, the underlying (hardware) platform, untouched. This example thus illustrates the strengths of our approach: thanks to SKIVA's support for aggregated bitslice operations, we benefit from techniques and advances in the field of hardware (*e.g.*, boolean masking) as well as software (*e.g.*, temporal redundancy) protection mechanisms, while taking full advantage of the flexibility of software.

## 6 Conclusion

We have presented SKIVA, a general-purpose 32-bit processor supporting high-throughput, secure block ciphers on embedded devices. Our objective in extending the SPARC instruction set was to provide cryptographers with a manageable programming model for implementing secure ciphers on a general-purpose CPU. On the software side, we advocate an approach centered around bitslicing, where cryptographic primitives are treated as combinational circuits. By design, bitslicing protects an implementation against timing-based side-channel attacks. But it also provides a sound basis for modular protections against fault and/or power-based side-channel attacks, thus paving the way for a pay-as-you-go security approach. In essence, SKIVA can be understood as a Turing machine for efficiently and securely executing combinational circuits in software.

These design choices translate into protection mechanisms that can naturally and systematically be integrated together. To protect against faults, we have shown that intra-instruction redundancy enables a purely analytic security analysis, guaranteeing significant coverage, while we experimental showed that temporal redundancy protects against instruction skips. To protect against side-channel, we crucially rely on the physical isolation of slices thus significantly reducing the risk of involuntary interference due to architectural details invisible to the programmer.

We have demonstrated the benefits of our approach with a bitsliced implementation of AES with 1, 2 and 4 shares, a temporal redundancy of 1 and 2 as well as a spatial redundancy of 1, 2, and 4. In terms of code size, we have shown that all security levels can be implemented in less than 14048B. In terms of performance, we have seen that it scales well with protection levels, dividing the throughput by 163 with all protections enabled at their maximum ($D = 4, R_s = 4, R_t = 2$).

**Future work.** In this paper, we have studied AES running on the SKIVA platform. To demonstrate the versatility of SKIVA, we intend to evaluate more ciphers at various security levels on this platform, including physical fault injection. Besides, we would like to compare it with alternative platforms, including general-purpose processors – ARM Cortex, AVR, or RISC-V – as well as cryptographic extensions – namely XCrypto [MPP19]. To cover this design space, we plan to invest in automation, integrating our countermeasures into a bitslicing compiler [Por01, MDLM18].

## Acknowledgements

The SKIVA design was supported in part by the National Science Foundation Award 1617203, and the National Institute for Standards and Technology Award 70NANDBH17280.

## References

[ABMP13] Giovanni Agosta, Alessandro Barenghi, Massimo Maggi, and Gerardo Pelosi. Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2013.

[AVFM07] Frédéric Amiel, Karine Villegas, Benoit Feix, and Louis Marcel. Passive and active combined attacks: Combining fault attacks and side channel analysis. In *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*, pages 92–102, 2007.

[BBD$^+$16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129, 2016.

[BCD$^+$13] G C Becker, Jennifer Cooper, E. DeMulder, Gilbert Goodwill, Jules Jaffe, G. Kenworthy, T. Kouzminov, Andrew Leiserson, Mark E. Marson, Pankaj Rohatgi, and Sami Saab. Test vector leakage assessment (TVLA) methodology in practice. 2013.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, pages 16–29, 2004.

[BCR16] Thierno Barry, Damien Couroussé, and Bruno Robisson. Compilation of a countermeasure against instruction-skip fault attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC, Prague, Czech Republic, January 20, 2016*, pages 1–6, 2016.

[BDF$^+$17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 535–566, Cham, 2017. Springer International Publishing.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES, 2005.

[BGG$^+$14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, pages 64–81, 2014.

[BGK05] Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably secure masking of aes. In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography*, pages 69–83, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[BGR18] Sonia Belaïd, Dahmun Goudarzi, and Matthieu Rivain. Tight private circuits: Achieving probing security with the least refreshing. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory* *and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, pages 343–372, 2018.

[BGRV15] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. Dpa, bitslicing and masking at 1 ghz. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 599–619, 2015.

[BGV11] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011*, pages 105–114, 2011.

[Bih97] Eli Biham. A fast new DES implementation in software. In *Fast Software Encryption (FSE)*, 1997.

[BJHB18] Jakub Breier, Dirmanto Jap, Xiaolu Hou, and Shivam Bhasin. On side-channel vulnerabilities of bit permutations: Key recovery and reverse engineering. *IACR Cryptology ePrint Archive*, 2018:219, 2018.

[BMP$^+$11] Manuel Barbosa, Andrew Moss, Dan Page, Nuno F. Rodrigues, and Paulo F. Silva. Type checking cryptography implementations. In *Fundamentals of Software Engineering - 4th IPM International Conference, FSEN 2011, Tehran, Iran, April 20-22, 2011, Revised Selected Papers*, pages 316–334, 2011.

[BRN$^+$15] Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. Automatic application of power analysis countermeasures. *IEEE Trans. Computers*, 64(2):329–341, 2015.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, pages 513–525, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[CBD$^+$15] Clement Champeix, Nicolas Borrel, Jean-Max Dutertre, Bruno Robisson, Mathieu Lisart, and Alexandre Sarafianos. Experimental validation of a bulk built-in current sensor for detecting laser-induced currents. In *21st IEEE International On-Line Testing Symposium, IOLTS 2015, Halkidiki, Greece, July 6-8, 2015*, pages 150–155, 2015.

[Col19] Laser-induced Single-bit Faults in Flash Memory: Instructions Corruption on a 32-bit Microcontroller. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019.

[CPM$^+$18] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. Screaming channels: When electromagnetic side channels meet radio transceivers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 163–177, 2018.

[CPT17] Lucian Cojocar, Kostas Papagiannopoulos, and Niek Timmers. Instruction duplication: Leaky and not too fault-tolerant! In *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, pages 160–179, 2017.

[CS18] Gaetan Cassiers and François-Xavier Standaert. Improved bitslice masking: from optimized non-interference to probe isolation. *IACR Cryptology ePrint Archive*, 2018:438, 2018.

[CSN$^+$17] Zhi Chen, Junjie Shen, Alex Nicolau, Alexander V. Veidenbaum, Nahid Farhady Ghalaty, and Rosario Cammarota. CAMFAS: A compiler approach to mitigate fault attacks via enhanced SIMDization. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017*, pages 57–64, 2017.

[dCKC$^+$16] Ruan de Clercq, Ronald De Keulenaer, Bart Coppens, Bohan Yang, Pieter Maene, Koen De Bosschere, Bart Preneel, Bjorn De Sutter, and Ingrid Verbauwhede. SOFIA: software and control flow integrity architecture. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 1172–1177, 2016.

[DEK$^+$18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: exploiting ineffective fault inductions on symmetric cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):547–572, 2018.

[DMM$^+$13] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria. Electromagnetic glitch on the AES round counter. In *Constructive Side-Channel Analysis and Secure Design - 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers*, pages 17–31, 2013.

[DPA00] Joan Daemen, Michaël Peeters, and Gilles Van Assche. Bitslice ciphers and power analysis attacks. In *Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings*, pages 134–149, 2000.

[DV12] François Dassance and Alexandre Venelli. Combined fault and side-channel attacks on the AES key schedule. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*, pages 63–71, 2012.

[EW14] Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In *International Conference on Computer Aided Verification*, pages 114–130. Springer, 2014.

[FJLT13] Thomas Fuhr, Eliane Jaulmes, Victor LomnÃĺ, and Adrian Thillard. Fault Attacks on AES with Faulty Ciphertexts Only. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 108–118, Aug 2013.

[GGP08] Philipp Grabher, Johann Großschädl, and Dan Page. Light-weight instruction set extensions for bit-sliced cryptography. In *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, pages 331–345, 2008.

[GJJR11] Gilbert Goodwill, Benjamin Jun, John Jaffe, and Pankaj Rohatgi. A testing methodology for side channel resistance. 2011.

[GJRS18] Dahmun Goudarzi, Anthony Journault, Matthieu Rivain, and François-Xavier Standaert. Secure multiplication for bitslice higher-order masking: Optimisation and comparison. In *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, pages 3–22, 2018.

[GM10] Berndt M. Gammel and Stefan Mangard. On the duality of probing and fault attacks. *J. Electronic Testing*, 26(4):483–493, 2010.

[GMK12] Xiaofei Guo, Debdeep Mukhopadhyay, and Ramesh Karri. Provably secure concurrent error detection against differential fault analysis. *IACR Cryptology ePrint Archive*, 2012:552, 2012.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, page 3, 2016.

[GPSS18] Benjamin Grégoire, Kostas Papagiannopoulos, Peter Schwabe, and Ko Stoffelen. Vectorizing higher-order masking. In *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, pages 23–43, 2018.

[GYCH16] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Cryptology ePrint Archive, Report 2016/613, 2016. https://eprint.iacr.org/2016/613.

[IL07] Paolo Ienne and Rainer Leupers. *Customizable Embedded Processors: Design Technologies and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[JS17] Anthony Journault and François-Xavier Standaert. Very high order masking: Efficient implementation and security evaluation. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 623–643, 2017.

[KDK$^+$14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372, 2014.

[Kön08] Robert Könighofer. A fast and cache-timing resistant implementation of the AES. In *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*, pages 187–202, 2008.

[KS09] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. *CHES*, 2009.

[LAZ$^+$17] Pei Luo, Konstantinos Athanasiou, Liwei Zhang, Zhen Hang Jiang, Yunsi Fei, A. Adam Ding, and Thomas Wahl. Compiler-assisted Threshold Implementation against Power Analysis Attacks. pages 541–544. IEEE, November 2017.

[LCFS18] Benjamin Lac, Anne Canteaut, Jacques J. A. Fournier, and Renaud Sirdey. Thwarting fault attacks against lightweight cryptography using SIMD instructions. In *IEEE International Symposium on Circuits and Systems, ISCAS 2018, 27-30 May 2018, Florence, Italy*, pages 1–5, 2018.

[LHB14] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. Software countermeasures for control flow integrity of smart card C codes. In *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*, pages 200–218, 2014.

[LLF16] Pei Luo, Chao Luo, and Yunsi Fei. System clock and power supply cross-checking for glitch detection. *IACR Cryptology ePrint Archive*, 2016:968, 2016.

[LSG$^+$10] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. Fault sensitivity analysis. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, pages 320–334, 2010.

[MDH$^+$13] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Encrenaz Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 77–88, 2013.

[MDLM18] Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne, and Gilles Muller. Usuba: Optimizing & trustworthy bitslicing compiler. In *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing, WPMVP@PPoPP 2018, Vienna, Austria, February 24, 2018*, pages 4:1–4:8, 2018.

[MOPT11] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Automatic insertion of DPA countermeasures. *IACR Cryptology ePrint Archive*, 2011:412, 2011.

[MOPT12] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, pages 58–75, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[MPP19] B. Marshall, D. Page, and T. Pham. XCrypto: a cryptographic ISE for RISC-V, 2019.

[NHH$^+$16] Shoei Nashimoto, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji, and Takafumi Aoki. Buffer overflow attack with multiple fault injection and a proven countermeasure. *Journal of Cryptographic Engineering*, 2016.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, pages 529–545, 2006.

[PH12] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

[PHBC17] Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. Compiler-assisted loop hardening against fault attacks. *ACM Trans. Archit. Code Optim.*, 14(4):36:1–36:25, December 2017.

[Por] Thomas Pornin. BearSSL, a smaller SSL/TLS library. https://bearssl.org. Accessed: 2019-01-08.

[Por01] Thomas Pornin. *Implantation et optimisation des primitives cryptographiques.* PhD thesis, École Normale Supérieure, 2001.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, pages 282–297, 2017.

[PYGS16] Conor Patrick, Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. Lightweight fault attack resistance in software using intra-instruction redundancy. In *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, pages 231–244, 2016.

[RBIK12] Francesco Regazzoni, Luca Breveglieri, Paolo Ienne, and Israel Koren. Interaction between fault attack countermeasures and the resistance against power analysis attacks. In *Fault Analysis in Cryptography*, pages 257–272. 2012.

[RBV17] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 1697–1702, 2017.

[RCS+09] Francesco Regazzoni, Alessandro Cevrero, François-Xavier Standaert, Stéphane Badel, Theo Kluter, Philip Brisk, Yusuf Leblebici, and Paolo Ienne. A design flow and evaluation framework for dpa-resistant instruction set extensions. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pages 205–219, 2009.

[RDB+18] Oscar Reparaz, Lauren De Meyer, Begül Bilgin, Victor Arribas, Svetla Nikova, Ventzislav Nikov, and Nigel P. Smart. CAPA: the spirit of beaver against physical attacks. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, pages 121–151, 2018.

[Res18] Cobham Gaisler Research. Leon-3 processor, 2018. https://www.gaisler.com/index.php/products/processors/leon3.

[RGV17] Oscar Reparaz, Benedikt Gierlichs, and Ingrid Verbauwhede. Fast leakage assessment. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 387–399, Cham, 2017. Springer International Publishing.

[RLK11] Thomas Roche, Victor Lomné, and Karim Khalfallah. Combined fault and side-channel attack on protected implementations of AES. In *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*, pages 65–83, 2011.

[RNR+15] Lionrl Rivière, Zakaria Najm, Pablo Rauzy, Jean-Luc. Danger, Julien Bringer, and Lionel Sauvage. High precision fault injections on the instruction cache of armv7-m architectures. In *IEEE International Symposium on Hardware Oriented Security and Trust, (HOST)*, pages 62–67, 2015.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, pages 413–427, 2010.

[RSD06] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. Bitslice implementation of AES. In *Cryptology and Network Security, 5th International Conference, CANS 2006, Suzhou, China, December 8-10, 2006, Proceedings*, pages 203–212, 2006.

[SBD$^+$18] Thierry Simon, Lejla Batina, Joan Daemen, Vincent Grosso, Pedro Maat Costa Massolino, Kostas Papagiannopoulos, Francesco Regazzoni, and Niels Samwel. Towards lightweight cryptographic primitives with built-in fault-detection. *IACR Cryptology ePrint Archive*, 2018:729, 2018.

[SI92] CORPORATE SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[SM15a] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 495–513, 2015.

[SM15b] Tobias Schneider and Amir Moradi. Leakage assessment methodology - a clear roadmap for side-channel evaluations. In *IACR Cryptology ePrint Archive*, 2015.

[SMG16] Tobias Schneider, Amir Moradi, and Tim Güneysu. Parti: Towards combined hardware countermeasures against side-channeland fault-injection attacks. In *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, page 39, 2016.

[SMKLM02] Yen Sung-Ming, Seungjoo Kim, Seongan Lim, and Sangjae Moon. A countermeasure against one physical cryptanalysis may benefit another attack. In Kwangjo Kim, editor, *Information Security and Cryptology — ICISC 2001*, pages 414–427, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[SS16] Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, pages 180–194, 2016.

[TG07] Stefan Tillich and Johann Großschädl. Power analysis resistant AES implementation with instruction set extensions. In *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, pages 303–319, 2007.

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In Claudio A. Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, pages 224–233, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Tri03] Elena Trichina. Combinational logic design for AES subbyte transformation on masked data. *IACR Cryptology ePrint Archive*, 2003:236, 2003.

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 1057–1074, 2017.

[TV04] Kris Tiri and Ingrid Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, pages 246–251, 2004.

[WWM15] Mario Werner, Erich Wenger, and Stefan Mangard. Protecting the control flow of embedded processors against fault attacks. In *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, pages 161–176, 2015.

[YGD+16] Bilgiday Yuce, Nahid F Ghalaty, Chinmay Deshpande, Conor Patrick, Leyla Nazhandali, and Patrick Schaumont. FAME: Fault-attack aware microprocessor extensions for hardware fault detection and software fault response. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, page 8. ACM, 2016.

[YSW18] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault attacks on secure embedded software: Threats, design, and evaluation. *Journal of Hardware and Systems Security*, 2(2):111–130, Jun 2018.

[YYP+18] Yuan Yao, Mo Yang, Conor Patrick, Bilgiday Yuce, and Patrick Schaumont. Fault-assisted side-channel analysis of masked implementations. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2018, Washington, DC, USA, April 30 - May 4, 2018*, pages 57–64, 2018.

[ZDCT13] Loïc Zussa, Jean-Max Dutertre, Jessy Clédière, and Assia Tria. Power supply glitch induced faults on FPGA: an in-depth analysis of the injection mechanism. In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS), Chania, Crete, Greece, July 8-10, 2013*, pages 110–115, 2013.

## A Custom instructions details

### A.1 TR2 instruction

```
TR2 rs1, rs2, rd

    $reg_{rd}$[31:0] := CONCAT($reg_{rs1}$[15],$reg_{rs2}$[15],$reg_{rs1}$[14],$reg_{rs2}$[14], ...
        $reg_{rs1}$[13],$reg_{rs2}$[13],$reg_{rs1}$[12],$reg_{rs2}$[12], ...
        $reg_{rs1}$[11],$reg_{rs2}$[11],$reg_{rs1}$[10],$reg_{rs2}$[10], ...
        $reg_{rs1}$[9],$reg_{rs2}$[9],$reg_{rs1}$[8],$reg_{rs2}$[8], ...
        $reg_{rs1}$[7],$reg_{rs2}$[7],$reg_{rs1}$[6],$reg_{rs2}$[6], ...
        $reg_{rs1}$[5],$reg_{rs2}$[5],$reg_{rs1}$[4],$reg_{rs2}$[4], ...
        $reg_{rs1}$[3],$reg_{rs2}$[3],$reg_{rs1}$[2],$reg_{rs2}$[2], ...
        $reg_{rs1}$[1],$reg_{rs2}$[1],$reg_{rs1}$[0],$reg_{rs2}$[0])
    y[31:0] := CONCAT($reg_{rs1}$[31],$reg_{rs2}$[31],$reg_{rs1}$[30],$reg_{rs2}$[30], ...
        $reg_{rs1}$[29],$reg_{rs2}$[29],$reg_{rs1}$[28],$reg_{rs2}$[28], ...
        $reg_{rs1}$[27],$reg_{rs2}$[27],$reg_{rs1}$[26],$reg_{rs2}$[26], ...
        $reg_{rs1}$[25],$reg_{rs2}$[25],$reg_{rs1}$[24],$reg_{rs2}$[24], ...
        $reg_{rs1}$[23],$reg_{rs2}$[23],$reg_{rs1}$[22],$reg_{rs2}$[22], ...
        $reg_{rs1}$[21],$reg_{rs2}$[21],$reg_{rs1}$[20],$reg_{rs2}$[20], ...
        $reg_{rs1}$[19],$reg_{rs2}$[19],$reg_{rs1}$[18],$reg_{rs2}$[18], ...
        $reg_{rs1}$[17],$reg_{rs2}$[17],$reg_{rs1}$[16],$reg_{rs2}$[16])
```

## A.2 INVTR2 instruction

```
INVTR2 rs1, rs2, rd

    $reg_{rd}$[31:0] := CONCAT($reg_{rs1}$[30],$reg_{rs1}$[28],$reg_{rs1}$[26],$reg_{rs1}$[24], ...
        $reg_{rs1}$[22],$reg_{rs1}$[20],$reg_{rs1}$[18],$reg_{rs1}$[16], ...
        $reg_{rs1}$[14],$reg_{rs1}$[12],$reg_{rs1}$[10],$reg_{rs1}$[8], ...
        $reg_{rs1}$[6],$reg_{rs1}$[4],$reg_{rs1}$[2],$reg_{rs1}$[0], ...
        $reg_{rs2}$[30],$reg_{rs2}$[28],$reg_{rs2}$[26],$reg_{rs2}$[24], ...
        $reg_{rs2}$[22],$reg_{rs2}$[20],$reg_{rs2}$[18],$reg_{rs2}$[16], ...
        $reg_{rs2}$[14],$reg_{rs2}$[12],$reg_{rs2}$[10],$reg_{rs2}$[8], ...
        $reg_{rs2}$[6],$reg_{rs2}$[4],$reg_{rs2}$[2],$reg_{rs2}$[0])
    y[31:0] := CONCAT($reg_{rs1}$[31],$reg_{rs1}$[29],$reg_{rs1}$[27],$reg_{rs1}$[25], ...
        $reg_{rs1}$[23],$reg_{rs1}$[21],$reg_{rs1}$[19],$reg_{rs1}$[17], ...
        $reg_{rs1}$[15],$reg_{rs1}$[13],$reg_{rs1}$[11],$reg_{rs1}$[9], ...
        $reg_{rs1}$[7],$reg_{rs1}$[5],$reg_{rs1}$[3],$reg_{rs1}$[1], ...
        $reg_{rs2}$[31],$reg_{rs2}$[29],$reg_{rs2}$[27],$reg_{rs2}$[25], ...
        $reg_{rs2}$[23],$reg_{rs2}$[21],$reg_{rs2}$[19],$reg_{rs2}$[17], ...
        $reg_{rs2}$[15],$reg_{rs2}$[13],$reg_{rs2}$[11],$reg_{rs2}$[9], ...
        $reg_{rs2}$[7],$reg_{rs2}$[5],$reg_{rs2}$[3],$reg_{rs2}$[1])
```

## A.3 SUBROT instruction

```
SUBROT rs, imm, rd

    IF imm[2:0] = 010
        FOR i:=0:15
            j := 2*i
            $reg_{rd}$[j+1:j] := $reg_{rs}$[j:j+1]
        ENDFOR
    ELIF imm[2:0] = 100
        FOR i:=0:7
            j := 4*i
            $reg_{rd}$[j+3:j] := CONCAT($reg_{rs}$[j+2:j],$reg_{rs}$[j+3])
        ENDFOR
    FI
```

## A.4 RED instruction

```
RED rs, imm, rd

    IF imm[2:0] = 010
        $reg_{rd}$[15:0] := $reg_{rs}$[15:0]
        $reg_{rd}$[31:16] := $reg_{rs}$[15:0]
        y[15:0] := $reg_{rs}$[31:16]
        y[31:16] := $reg_{rs}$[31:16]
    ELIF imm[2:0] = 011
        $reg_{rd}$[15:0] := $reg_{rs}$[15:0]
        $reg_{rd}$[31:16] := (NOT $reg_{rs}$[15:0])
        y[15:0] := r$reg_{rs}$s[31:16]
        y[31:16] := (NOT $reg_{rs}$[31:16])
    ELIF imm[2:0] = 100
        $reg_{rd}$[7:0] := $reg_{rs}$[7:0]
        $reg_{rd}$[15:8] := $reg_{rs}$[7:0]
        $reg_{rd}$[23:16] := $reg_{rs}$[7:0]
        $reg_{rd}$[31:24] := $reg_{rs}$[7:0]
        y[7:0] := $reg_{rs}$[15:8]
        y[15:8] := $reg_{rs}$[15:8]
        y[23:16] := $reg_{rs}$[15:8]
        y[31:24] := $reg_{rs}$[15:8]
    ELIF imm[2:0] = 101
        $reg_{rd}$[7:0] := $reg_{rs}$[7:0]
        $reg_{rd}$[15:8] := (NOT $reg_{rs}$[7:0])
        $reg_{rd}$[23:16] := $reg_{rs}$[7:0]
        $reg_{rd}$[31:24] := (NOT $reg_{rs}$[7:0])
        y[7:0] := rs[15:8]
        y[15:8] := (NOT $reg_{rs}$[15:8])
        y[23:16] := rs[15:8]
        y[31:24] := (NOT $reg_{rs}$[15:8])
    ELIF imm[2:0] = 110
        $reg_{rd}$[7:0] := $reg_{rs}$[23:16]
        $reg_{rd}$[15:8] := $reg_{rs}$[23:16]
        $reg_{rd}$[23:16] := $reg_{rs}$[23:16]
        $reg_{rd}$[31:24] := $reg_{rs}$[23:16]
        y[7:0] := $reg_{rs}$[31:24]
        y[15:8] := $reg_{rs}$[31:24]
        y[23:16] := $reg_{rs}$[31:24]
        y[31:24] := $reg_{rs}$[31:24]
    ELIF imm[2:0] = 111
        $reg_{rd}$[7:0] := $reg_{rs}$[23:16]
        $reg_{rd}$[15:8] := (NOT $reg_{rs}$[23:16])
        $reg_{rd}$[23:16] := $reg_{rs}$[23:16]
        $reg_{rd}$[31:24] := (NOT $reg_{rs}$[23:16])
        y[7:0] := $reg_{rs}$[31:24]
        y[15:8] := (NOT $reg_{rs}$[31:24])
        y[23:16] := $reg_{rs}$[31:24]
        y[31:24] := (NOT $reg_{rs}$[31:24])
    FI
```

## A.5 ANDC16 instruction

```
ANDC16 rs1, rs2, rd

        $reg_{rd}$[15:0] := ($reg_{rs1}$[15:0] AND $reg_{rs2}$[15:0])
        $reg_{rd}$[31:16] := ($reg_{rs1}$[31:16] OR $reg_{rs2}$[31:16])
```

## A.6 XORC16 instruction

```
XORC16 rs1, rs2, rd

        $reg_{rd}$[15:0] := ($reg_{rs1}$[15:0] XOR $reg_{rs2}$[15:0])
        $reg_{rd}$[31:16] := ($reg_{rs1}$[31:16] XNOR $reg_{rs2}$[31:16])
```

## A.7 XNORC16 instruction

```
XNORC16 rs1, rs2, rd

        $reg_{rd}$[15:0] := ($reg_{rs1}$[15:0] XNOR $reg_{rs2}$[15:0])
        $reg_{rd}$[31:16] := ($reg_{rs1}$[31:16] XOR $reg_{rs2}$[31:16])
```

## A.8 ANDC8 instruction

```
ANDC8 rs1, rs2, rd

        $reg_{rd}$[7:0] := ($reg_{rs1}$[7:0] AND $reg_{rs2}$[7:0])
        $reg_{rd}$[15:8] := ($reg_{rs1}$[15:8] OR $reg_{rs2}$[15:8])
        $reg_{rd}$[23:16] := ($reg_{rs1}$[23:16] AND $reg_{rs2}$[23:16])
        $reg_{rd}$[31:24] := ($reg_{rs1}$[31:24] OR $reg_{rs2}$[31:24])
```

## A.9 XORC8 instruction

```
XORC8 rs1, rs2, rd

        $reg_{rd}$[7:0] := ($reg_{rs1}$[7:0] XOR $reg_{rs2}$[7:0])
        $reg_{rd}$[15:8] := ($reg_{rs1}$[15:8] XNOR $reg_{rs2}$[15:8])
        $reg_{rd}$[23:16] := ($reg_{rs1}$[23:16] XOR $reg_{rs2}$[23:16])
        $reg_{rd}$[31:24] := ($reg_{rs1}$[31:24] XNOR $reg_{rs2}$[31:24])
```

## A.10 XNORC8 instruction

```
XNORC8 rs1, rs2, rd

        $reg_{rd}$[7:0] := ($reg_{rs1}$[7:0] XNOR $reg_{rs2}$[7:0])
        $reg_{rd}$[15:8] := ($reg_{rs1}$[15:8] XOR $reg_{rs2}$[15:8])
        $reg_{rd}$[23:16] := ($reg_{rs1}$[23:16] XNOR $reg_{rs2}$[23:16])
        $reg_{rd}$[31:24] := ($reg_{rs1}$[31:24] XOR $reg_{rs2}$[31:24])
```

## A.11 FTCHK instruction

```
FTCHK rs, imm, rd

    IF imm[3:0] = 1010
        FOR i:=0:15
            $reg_{rd}$[i] := ($reg_{rs}$[i+16] XOR $reg_{rs}$[i])
            $reg_{rd}$[i+16] :=  ($reg_{rs}$[i+16] XNOR $reg_{rs}$[i])
        ENDFOR
    ELIF imm[3:0] = 1011
```

```
        FOR i:=0:15
            reg$_{rd}$[i] := (reg$_{rs}$[i+16] XNOR reg$_{rs}$[i])
            reg$_{rd}$[i+16] :=  (reg$_{rs}$[i+16] XOR reg$_{rs}$[i])
        ENDFOR
    ELIF imm[3:0] = 1100
        FOR i:=0:7
            reg$_{rd}$[i] := ((reg$_{rs}$[i+8] XOR reg$_{rs}$[i]) OR ...
                            (reg$_{rs}$[i+16] XOR reg$_{rs}$[i]) OR ...
                            (reg$_{rs}$[i+24] XOR reg$_{rs}$[i]))
            reg$_{rd}$[i+8] := ((reg$_{rs}$[i+8] XNOR reg$_{rs}$[i]) AND ...
                            (reg$_{rs}$[i+16] XNOR reg$_{rs}$[i]) AND ...
                            (reg$_{rs}$[i+24] XNOR reg$_{rs}$[i]))
            reg$_{rd}$[i+16] := ((reg$_{rs}$[i+8] XOR reg$_{rs}$[i]) OR ...
                            (reg$_{rs}$[i+16] XOR reg$_{rs}$[i]) OR ...
                            (reg$_{rs}$[i+24] XOR reg$_{rs}$[i]))
            reg$_{rd}$[i+24] := ((reg$_{rs}$[i+8] XNOR reg$_{rs}$[i]) AND ...
                            (reg$_{rs}$[i+16] XNOR reg$_{rs}$[i]) AND ...
                            (reg$_{rs}$[i+24] XNOR reg$_{rs}$[i]))
        ENDFOR
    ELIF imm[3:0] = 1101
        FOR i:=0:7
            reg$_{rd}$[i] := ((reg$_{rs}$[i+8] XNOR reg$_{rs}$[i]) OR ...
                            (reg$_{rs}$[i+16] XOR reg$_{rs}$[i]) OR ...
                            (reg$_{rs}$[i+24] XNOR reg$_{rs}$[i]))
            reg$_{rd}$[i+8] := ((reg$_{rs}$[i+8] XOR reg$_{rs}$[i]) AND ...
                            (reg$_{rs}$[i+16] XNOR reg$_{rs}$[i]) AND ...
                            (reg$_{rs}$[i+24] XOR reg$_{rs}$[i]))
            reg$_{rd}$[i+16] := ((reg$_{rs}$[i+8] XNOR reg$_{rs}$[i]) OR ...
                            (reg$_{rs}$[i+16] XOR reg$_{rs}$[i]) OR ...
                            (reg$_{rs}$[i+24] XNOR reg$_{rs}$[i]))
            reg$_{rd}$[i+24] := ((reg$_{rs}$[i+8] XOR reg$_{rs}$[i]) AND ...
                            (reg$_{rs}$[i+16] XNOR reg$_{rs}$[i]) AND ...
                            (reg$_{rs}$[i+24] XOR reg$_{rs}$[i]))
        ENDFOR
    FI
```

## B Efficient C emulation of the custom instructions

We provide here the C codes for emulating some of the custom instructions. We omitted `ftchk`, `red`, `tr2` and `invtr2`, for which the emulation code is the straightforward implementation of the specification.

```
#define ANDC8(r,a,b)   r = (((a) | (b)) & 0xFF00FF00) | (((a) & (b)) & 0x00FF00FF)
#define XORC8(r,a,b)   r = (a) ^ (b) ^ 0xFF00FF00
#define XNORC8(r,a,b)  r = (a) ^ (b) ^ 0x00FF00FF
#define ANDC16(r,a,b)  r = (((a) | (b)) & 0xFFFF0000) | (((a) & (b)) & 0x0000FFFF)
#define XORC16(r,a,b)  r = (a) ^ (b) ^ 0xFFFF0000
#define XNORC16(r,a,b) r = (a) ^ (b) ^ 0x0000FFFF
```

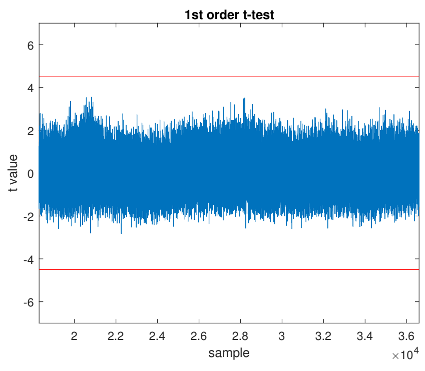
## C Side-channel analysis results

### C.1 CPA results

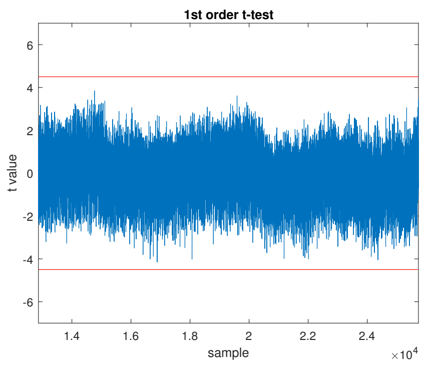
**Table 4:** Detailed report of 1st order CPA results on unmasked SubBytes of 1st round AES

| # of traces | # of key bytes revealed |
|---|---|
| 3K | 1 |
| 4K | 3 |
| 9K | 5 |
| 10K | 6 |
| 11K | 7 |
| 12K | 8 (half key) |
| 14K | 10 |
| 18K | 11 |
| 19K | 12 |
| 21K | 13 |
| 22K | 14 |
| 23K | 15 |
| 24K | 16 (full key) |

### C.2 TVLA results



**(a)** Complementary redundancy



**(b)** Direct redundancy

**Figure 11:** 1st order t-test on 1st order masked AES S-box in complementary and direct redundancy (25K fixed vs. 25K random)