Lightweight Fault Attack Resistance in Software Using Intra-Instruction Redundancy

Conor Patrick, Bilgiday Yuce, Nahid Farhady Ghalaty, Patrick Schaumont

Bradley Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, USA
{conorpp,bilgiday,farhady,schaum}@vt.edu

Abstract. Fault attack countermeasures can be implemented by storing or computing sensitive data in redundant form, such that the faulty data can be detected and restored. We present a class of lightweight, portable software countermeasures for block ciphers. Our technique is based on redundant bit-slicing, and it is able to detect faults in the execution of a single instruction. In comparison to earlier techniques, we are able to intercept data faults as well as instruction sequence faults using a uniform technique. Our countermeasure thwarts precise bit-fault injections through pseudo-random shifts in the allocation of data bit-slices. We demonstrate our solution on a full AES design and confirm the claimed security protection through a detailed fault simulation for a 32-bit embedded processor. We also quantify the overhead of the proposed fault countermeasure, and find a minimal increase in footprint (14%), and a moderate performance overhead between 125% to 317%, depending on the desired level of fault-attack resistance.

Keywords: Fault attacks, Fault resistance, Intra-instruction redundancy, Bitslicing, Block ciphers

1 Introduction

The injection of faults in cryptographic software is a well-studied technique to extract cryptographic keys. Originally demonstrated against public-key cryptography [1], their scope has since been widened to the symmetric-key case. The current state of the art in differential fault analysis on the Advanced Encryption Standard can extract an AES-128 key with just two faults [2]. Therefore, for applications where fault injection is a relevant threat, it is crucial to detect the occurrence of even a single fault and respond appropriately. In this contribution, we study and develop countermeasure techniques that are applicable to software, and that does not need any special hardware. Software countermeasures against fault attacks are commonly developed using redundancy. However, all redundancy based techniques share a common weakness: they are ineffective against an adversary who can inject consistent faults in redundant sections of code or data. This is especially relevant for implementations that are time-redundant, since they only require the adversary to inject the same fault sequentially.

In this paper, we propose a technique that enables the software to exploit redundancy for fault attack protection *within* a single instruction; we propose the term *intra-instruction redundancy* to describe it. Our technique is cross-platform and requires that an algorithm be bit-sliced. We focus on block ciphers, as they all can be bit-sliced. To protect from computation faults, we allocate some bit-slices as redundant copies of the true payload data slices. A data fault can then be detected by the difference between a data slice and its redundant copy after the encryption of the block completes. To protect the computations from instruction faults such as instruction skip, we also allocate some of the bit-slices as check-slices which compute a known result. The *intra-instruction redundancy* countermeasure is thus obtained through the bit-sliced design of a cipher, with redundant data-slices to detect computation faults and check-slices to detect instruction faults. This basic mechanism is then further strengthened against targeted fault-injection as follows. First, we pipeline the bit-sliced computation such that each slice computes a different encryption round. Since a fault-injection adversary is typically interested in the last or penultimate round, and since there will be only a few bits in a word that contain such a round, the *pipelined intra-instruction redundancy* countermeasure reduces the attack surface considerably. Second, we also randomize the slice assignment after each encryption, such that a cipher round never remains on a single slice for more than a single encryption. We show that this final countermeasure, the *shuffled pipelined intra-instruction redundancy*, is very effective and requires an adversary who can control fault injection with single-cycle, bit-level targeting chosen bits. We are not aware of a fault injection mechanism that achieves this level of precision.

The contributions of the paper are as follows.

- We propose a software countermeasure based on redundant bit slicing. The bit slices are used for data redundancy as well as control redundancy. The latter is achieved by computing a known answer.
- The proposed countermeasure is generic and can still be used in combination with other software countermeasures such as infective countermeasures or side-channel resistant techniques based on masking.
- The security of the proposed countermeasure is quantitatively analyzed to establish estimated fault coverage. In addition, it is empirically tested using simulation for different fault models including instruction skip, random word, random byte and bit-precision faults.
- The bit-sliced design leads to a secure fault detection and fault handling approach that is purely computational, and that avoids comparison and decision making. This avoids a well-known single point-of-failure in redundancy-based countermeasures.
- We evaluate the overhead of the countermeasure over an unprotected, bit-sliced implementation of AES-128 that runs at 469.3 cycles/bytes, we show that the highest level of protection is achieved at 1957 cycles per byte, which protects against targeted, repeatable, multiple bit faults.

The rest of the paper is organized as follows. In the next section, we provide additional details on the fault models used in this work. In Section III, we highlight the differences of previous software countermeasures with our proposed countermeasure. Section IV is an up-close discussion of the design of our countermeasure; we elaborate on the bit-slice allocation strategy and on the integration of the protected design on embedded platforms. Section V estimates the fault coverage of the proposed countermeasures under different fault models. Section VI presents the implementation overhead for a 32-bit embedded processor and empirically demonstrates the fault countermeasure operation using fault simulation. We conclude the paper in Section VII.

2 Fault Models

This section details the fault models that we used in this paper to evaluate our countermeasures. The fault model is the expected effect of the fault injection on a cryptosystem. The manipulated data may affect instruction opcodes as well as data, and we distinguish these two cases as instruction faults and computation faults.

Computation Faults: These faults cause errors in the data that is processed by a program. There is a trade-off between the accuracy by which an adversary can control the fault injection, and the required sophistication of a fault countermeasure that thwarts it. Therefore, we assume four computational fault models:

1. *Random Word*: The adversary can target a specific word in a program and change its value into a random value unknown to the adversary.
2. *Random Byte*: The adversary can target a specific word in a program and change a single byte of it into a random value unknown to the adversary.
3. *Random Bit*: The adversary can target a specific word in a program and change a single bit of it into a random value unknown to the adversary.
4. *Chosen Bit Pair*: The adversary can target a chosen bit pair of a specific word in a program, and change it into a random value unknown to the adversary.

Instruction Faults: This fault model assumes that an attacker can change the opcode of an instruction by fault injection. A very common model is the *Instruction Skip* fault model, which replaces the opcode of an instruction with a `nop` instruction. Using this model, an attacker can skip the execution of a specific instruction in the program.

3 Related Work

The two principal techniques for a fault countermeasure are detection-based and infection-based [3]. We start with detection-based countermeasures in software, as they are most similar to our proposal. A classic technique relies on time redundancy, such as duplicate encryption or encryption followed by decryption. This allows the detection of faults by comparing the consistency of the redundant executions. These techniques, however, do not work well against an adversary who is able to inject consistent faults in the redundant copies, or against an adversary who directly targets the comparison. Several time redundant techniques have been proposed to make consistent fault injection more difficult. For example, instruction duplication and triplication [4] are used because it is assumed that back-to-back fault injection is harder than fault injections that are relatively far spaced apart. Duplication and triplication were found to incur 3.4 and 10.6 times performance overhead, respectively [4]. However, duplication and triplication are relatively easy to overcome with a modern fault injection setup. More sophisticated techniques are possible, but they are algorithm-specific. Examples are techniques based on invariant properties of a block cipher [5], or based on storing sensitive variables in a transformed format [6]. However, we are interested in a generic, algorithm independent technique.

Another category of detection-based countermeasures use information redundancy, which uses additional check variables or parity bits [4] to detect faults in the data. This was found to incur between 3.5 - 4.7 times performance overhead [4]. A recent proposal observed that Wave Dynamic Differential Logic (WDDL), which represents data in complementary format, is able to detect computation faults [7] but no performance metrics are provided. While these information-based countermeasures are generic and easy to apply to a broad class of algorithms, they are unable to detect low-level instruction-level faults in the underlying processor when implemented in software.

The second major class of countermeasures uses infection. The idea is that injected faults will also destroy the invariant properties of the fault. This effectively eliminates the possibility of differential fault analysis. However, for every infective countermeasure proposed so far, a corresponding attack has been demonstrated [8, 9].

Most related works show that they have good fault coverage but it's under a narrow fault model. A common fault model is to assume that attackers can only inject one fault at a time. But if an attacker can inject more than one fault or affect multiple instructions with one fault, the fault coverage is likely to plummet.

In this paper, we propose detection-based countermeasures against fault attacks in software that are based on intra-instruction redundancy. We go beyond redundant encoding of information by also including the ability to detect instruction faults as well as computation faults. We show that these countermeasures can protect against a variety of realistic fault models. To the best of our knowledge, this is the first work that provides comprehensive coverage against processor-level fault attacks.

4 Proposed Software Countermeasures for Fault Attacks

We will explain the motivation and main idea of our countermeasures. We will then explain how they can be implemented. Finally, we will provide some discussion on the performance and footprint impact.

The countermeasures are based on bit-slicing. As we will explain further, bit-slicing allows for a program to dynamically select different data flows to be present in a processor word. This is attractive for a fault attack countermeasure because it presents a new way to leverage redundancy. Each data word can be split amongst regular data streams and redundant data streams, allowing redundancy to be

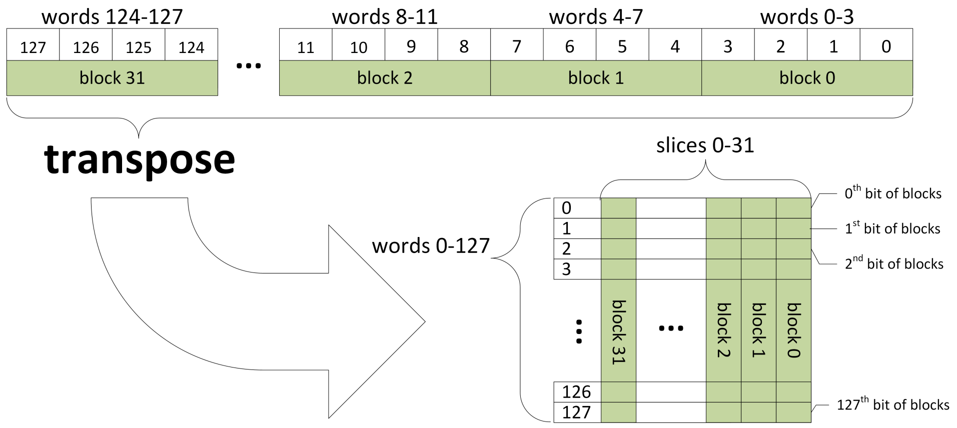

Fig. 1. Transpose of 32 blocks to fit bitwise into 128 32-bit words.

present spatially in all instructions without actually having to re-execute anything. Because the redundancy is interleaved with the data in every instruction, we call this Intra-Instruction Redundancy (IIR). By never separating the data from the redundancy in the processor word, we use pure spatial redundancy rather than commonly used time-based redundancy, which is vulnerable to repeated fault injections [10].

In this work we consider two ways to detect faults using redundancy. First, if you are computing data where the result is unknown, you can only detect a fault by recomputing the data an additional time to compare the results. Second, if the result is already known before computation, you can store a read only copy of the result and only need to execute on the data once to reproduce the result and check that it is the same.

Our countermeasure scheme relies on making comparisons at the end of encryption rounds. Because the comparisons are a very small part of the code, we assume we can cheaply duplicate them enough such that an adversary may not reasonably skip all of them using faults. In the advent of a fault detection, a random cipher text is output and the program may either restart encryption or enact a different, application-specific policy.

4.1 Bit-slicing without Fault Attack Protection

Bit-slicing is a technique used commonly in block ciphers and embedded systems to fully utilize the word length of a processor for all operations, potentially increasing the total throughput. Bit-slicing avoids data-dependent memory lookups and because of that, data-dependent cache effects. It involves decomposing all components into boolean operations and orienting the data such that one bit can be computed at a time per instruction. If one bit is computed at a time, then a 32 bit processor word can be filled with 32 different blocks, computing all blocks simultaneously.

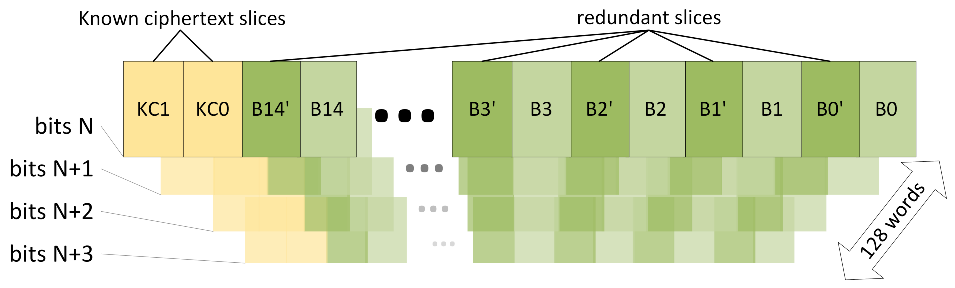

Fig. 2. Bit-slicing with Intra-Instruction Redundancy using 15 data (B), 15 redundant (B'), and 2 known ciphertext (KC) slices. Each KC slice is aligned with its corresponding round key slices in other words.

A prerequisite for bit-slicing is to transpose the layout of input blocks, as shown in Figure 1. At the top, a traditional layout of blocks is depicted. There are 32 blocks, each composed of 4, 32 bit data words. All of them must be transposed. In the transposed layout, each word contains one bit from every block. In this format, each bit from 32 different blocks can be computed simultaneously for any instruction. A *slice* refers to a bit location in all words that together make up one block.

4.2 Intra-Instruction Redundancy

In traditional bit-sliced implementations, each slice is allocated to operate on a different input block for maximum throughput (Fig. 1). Instead, we separate slices into three categories: data slices (B), redundant slices (B'), and Known Ciphertext (KC) slices for fault detection (Fig. 2). Data slices and redundant slices operate on the same input plaintext, and thus, they produce the same ciphertext if no fault occurs. If a fault occurs during their execution, then it will be detected when results are compared at the end of encryption.

However, if both B and B' experience the same fault, then both of them will have the same faulty ciphertext and a fault cannot be detected. For example, this would always be the case for instruction skips. To address this issue, we include KC slices in addition to data and redundant slices. Instead of encrypting the input plaintexts with the run-time secret key, KC slices encrypt internally stored plaintexts with a different key, each of which are decided at design time. Therefore, the correct ciphertexts corresponding to these internally stored plaintexts are known by the software designer beforehand. If no fault is injected into the execution of a KC slice, it will produce a run-time ciphertext that is equal to the known, design-time ciphertext. In case of a computation or instruction fault, the run-time ciphertext will be different than the design-time ciphertext. Therefore, the run-time and design-time ciphertexts of the KC slices are compared at the end of encryption for fault detection. Because the round keys for the data slices and round keys for the KC slices can be intermixed at the slice level, we can execute KC slices together with the data and redundant slices. Figure 2 shows the slice

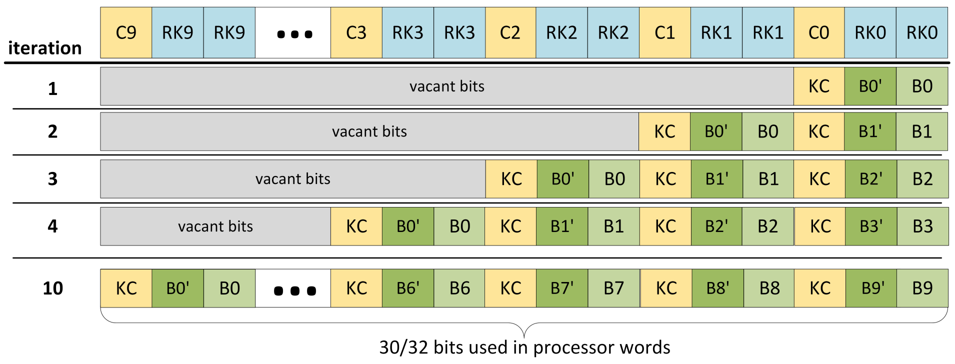

Fig. 3. Pipelined bit-sliced layout for 32 bit processor. $RK0 - 9$ are ten different round keys. $B0 - 9$ are different input blocks and $B'09$ are their redundant copies. KC is a known ciphertext slice. $C0 - 9$ are the round keys used to produce the known ciphertext.

allocation used in this work, which includes 15 data slices ($B0 - 14$), 15 redundant slices ($B'0 - 14$), and 2 KC slices ($KC0 - 1$). All slices are split across 128 words for a 128 bit block size.

A set of known plaintext-ciphertext pairs is included in the program from which KC slices can be selected from randomly for an encryption. This is because each KC slice only has a 50% chance of detecting an instruction fault. If only a couple of them are used, then there will likely be parts of the block cipher where instruction faults do not affect the KC slices. By selecting from a larger set of ciphertext-plaintext pairs, we significantly reduce the chance of an adversary finding such parts of the program. Each plaintext-ciphertext pair will be the size of two blocks of the cipher.

An adversary can bypass this countermeasure by injecting two bit faults that are next to each other in the processor word. The two bit fault has to align with any of the B slices and the corresponding B' slice. Then both will produce the same faulty ciphertext, going undetected.

4.3 Pipelined Intra-Instruction Redundancy

For an adversary to carry out a fault analysis attack, he must inject a fault into a target round of the block cipher [11, 12]. It is not enough to cause an undetected fault in the wrong round, as the faulty ciphertext will not be useful in analysis. Previously, all data and redundant slice pairs in the target word operate on the same round. Therefore, an adversary can target any combinations of these pairs to bypass IIR. Here we will explain how we can make the rounds spatial by making them correspond to slices within each word, instead of different words executed at different times. This makes fault injection harder as the faults will have to target specific bit locations.

Because block cipher rounds differ only in the round key used, we can make different bits correspond to different rounds by aligning slices with different round keys. Doing this means blocks will be computed in a pipelined fashion as shown in Figure 3, which shows ten rounds. The round keys are doubled and interleaved with the known ciphertext key beforehand to align with the pipeline. Each block is transposed one at a time rather than 32 at a time. For every iteration, 3 slices are shifted into the 128 word state (1 data, 1 redundant, and 1 KC). Initially shifted in is $B0$. Running for one iteration will compute round one of $B0$. Applying another shift aligns $B0$ for round 2 and shifts in $B1$ for round 1. This eliminates the need to have a set of plaintext-ciphertext pairs as it will be okay to have one pair. One pair will effectively make 10 different KC slices amongst the 10 rounds.

In this pipeline, because each set of 3 bits corresponds to a different round, any two bit fault will not suffice to undo the countermeasure. There is now only one valid bit location to successfully inject a 2 bit fault. For example, to fault round 9, a 2 bit fault must be injected at bit location 27. It is non-trivial for an adversary to inject a fault that is in a target bit location and consists of two adjacent bits.

Astute readers will point out that the last round in some block ciphers differs in more than just the round key. For example, in AES, the last round does not have the mix-columns step. Some additional masking can done to remove the effect of particular steps on any round(s). To be able to pipeline rounds that differ in steps, we add the following computation to each operation in the special step.

B = (BS & RM) | (B & ∼RM)

Where B is the block going through a particular step, BS is the computed result after the step, and RM is a mask representing the rounds that use the step. By doing this, the step will be applied to only the rounds that use it and leave the other round(s) the same.

4.4 Shuffled, Pipelined Intra-Instruction Redundancy

For our final countermeasure stage, we assume a highly skilled adversary who can inject multiple bit faults into target bit locations. In our case, we need to protect from a targeted 2-bit fault.

For each plaintext, we can effectively apply a random rotation to all of the slices and their corresponding round keys. The randomness is from an initial secret number that is continually XOR'd with generated ciphertext. We can reasonably assume that the adversary will not be able to predict the random rotation. Despite the adversary being able to inject known bit location faults, he will not know which bit corresponds to what round, making the attack much more difficult.

To support random shifts, we support dynamic allocation of each slice in the processor word, rather than statically defining which bits correspond to each round. The transpose step will have to support transposing into and out of any target bit location, rather than always shifting into bit location 0 and shifting out of bit location 29.

4.5 Secure, Comparison-Free Fault Handling

Rather than checking the memory using excessively duplicated comparisons, fault handling can be done in a purely computational approach. This approach is ideal because an application no longer needs to have a secure response to a fault injection.

If either a block, B, or its respective redundant slice, B', contain an error, we would expect the XOR of them $B \oplus B'$ to be nonzero. Whereas a non-faulty operation would always produce zero. Building upon this, we can make a method such that when a fault is injected, only a random number is output, foiling any attempt of fault analysis.

After encryption, we can compute the following mask.

MASK = (−(B ⊕ B') >> 128)

If B and B' are the same, then $B \oplus B'$ will be zero and the signed shift will move in all zeros. If $B \oplus B'$ is nonzero, then the signed shift will move in all ones. We can easily extend this mask to check a KC slice as well for instruction faults using our known ciphertext KC'.

MASK = (−(B ⊕ B') >> 128) | (−(KC ⊕ KC') >> 128)

As in our pipelined countermeasure, a redundant slice, data slice, and KC slice can be shifted out every iteration to compute the mask. We can then use this mask to protect our ciphertext block before it is output.

OUTPUT = (MASK & R) | (∼MASK & B);

By doing this, only our random number R will be output when a fault is detected. Otherwise, the correct ciphertext B will be output. Because these computations are not covered by intra-instruction redundancy, they would have to be duplicated using traditional approaches to protect from instruction faults. They are a small part of the code, so they can easily be duplicated without significantly increasing the footprint size. Computation faults need not be protected from as they would either cause $B \oplus B'$ or $KC \oplus KC'$ to be nonzero or just flip bits in the already computed ciphertext.

5 Security Analysis of the Proposed Countermeasures

In this section, we provide a security analysis for the proposed countermeasures in Section 4 against the fault models defined in Section 2.

Similar to Guo et al. [13], we use the Fault Coverage (FC) to quantify the security level of countermeasures. For a given countermeasure c and fault model f, we compute the fault coverage using Equation 1. In our computations, we assume that the adversary aims at injecting a computation or instruction skip fault into the execution of a target round.

$$(FC)_c^f = 1 - \frac{F_{undetected}}{F_{total}} \tag{1}$$

In Equation 1, F_{total} is the total number of faults covered by the fault model f. $F_{undetected}$ is the number of faults that affect the execution of the target round r, but cannot be detected by the given countermeasure c. More capable adversaries can increase the $F_{undetected}$, and reduce the F_{total} by accurately tuning fault injection parameters. We list our FC computations in Table 1.

Table 1. Theoretical Security Analysis of the Proposed Countermeasures

Countermeasure	**Computation Fault Models**				**Instruction Fault Models**
	Random Word	**Random Byte**	**Random Bit**	**Chosen Bit Pair**	**Instruction Skip**
Unprotected AES	0%	0%	0%	0%	0%
IIR-AES	$\approx$ 100%	94.90%	100%	51.61%	75%
Pipelined IIR-AES	$\approx$ 100%	99.90%	100%	96.77%	99.90%
Shuffled Pipelined IIR-AES	$\approx$ 100%	99.90%	100%	96.77%	99.90%

5.1 Security Analysis of Unprotected AES

In the unprotected, bit-sliced AES implementation, any computation or instruction fault during the execution of the target round r will be useful for the adversary. As there is no detection mechanism for this implementation, F_{total} and $F_{undetected}$ will be equal to each other. As a result, fault coverage will be 0 in any case. The detailed explanations for each fault model are as follows.

In the *Random Word* fault model, the adversary has no control on the number of faulty bits. The adversary can only create random faults in the target word (32-bit). For each fault injection, the difference between the correct word and the corresponding faulty word can have $(2^{32}-1)$ different values. Therefore, F_{total} and $F_{undetected}$ are $(2^{32}-1)$.

In the *Random Byte* fault model, an adversary can tune the fault injection to randomly affect a single byte of the 32-bit data. This adversary can inject a fault into one of the four bytes of the data. Each fault injection can create (2^8-1) different faults in a byte. As a result, F_{total} and $F_{undetected}$ are $4 \times (2^8-1)$.

In the *Random Bit* fault model, the fault injection can be tuned to affect single bit of the target word. Therefore, F_{total} and $F_{undetected}$ are 32.

In the *Chosen Bit Pair* fault model, the adversary can inject faults into two chosen, adjacent bits of the target word. Therefore, F_{total} and $F_{undetected}$ are 31.

5.2 Security Analysis of IIR-AES

To thwart this countermeasure, the adversary needs to create the same effect on the data slices and their corresponding redundant slices, without affecting any KC slice. Affecting any combination of 15 data and redundant slice pairs will create undetected faults. A *Random Word* fault can achieve this in $\sum_{i=1}^{15}\binom{15}{i}-\binom{15}{0}=(2^{15}-1)$ different ways. Therefore, $F_{undetected}$ is $(2^{15}-1)$ and the FC (using Eq. 1) is 99.9992% ($\approx$ 100%).

This countermeasure has three data and redundant slice pairs in the most significant byte of the target word, while it has four pairs in each of the remaining bytes (Fig. 2). Thus, for *Random Byte* fault model, the $F_{undetected}$ is $(2^3-1)+3\times(2^4-1)=52$, and the FC is 94.90%.

As a *Random Bit* fault can manipulate only a single KC slice, data slice, or redundant slice, the $F_{undetected}$ is 0, and the FC is 100%.

As a *Chosen Bit Pair* fault can target a specific pair of data and redundant slices, the $F_{undetected}$ is 15, and the FC is 51.61%.

An *Instruction skip* fault will have the same effect on a data slice and its corresponding redundant slice. Thus, data and redundant slice pairs cannot detect an instruction skip. Each KC slice has a 50% chance of detecting an instruction skip. As we have 2 KC slices, the fault coverage is $1 - \frac{1}{2^2} = 75\%$.

5.3 Security Analysis of Pipelined IIR-AES

In this countermeasure, the 32-bit word consists of 10 KC, 10 redundant, 10 data, and 2 spare slices (Fig. 3). Each data and redundant slice pair apply a different round of AES on a different block. As there is only one data and redundant slice pair running the target round r, the only way to obtain a useful and undetected computation fault is by targeting this pair of slices.

For *Random Word* and *Random Byte* faults, the $F_{undetected}$ is equal to 1. Therefore the corresponding fault coverage for Random Word and Random Byte faults are $\approx 100\%$ and 99.90%, respectively.

As no *Random Bit* fault can bypass this countermeasure, the $F_{undetected}$ is 0, the fault coverage is 100%.

A *Chosen Bit Pair* fault can manipulate the data and redundant slice pair that computes the target round r. The $F_{undetected}$ is 1 and the fault coverage is 96.77%.

This countermeasure significantly increases the fault coverage against *instruction skip* attacks as we use 10 constant slices. The only undetected instruction skip fault is the one that does not affect any of the constant bits. Therefore, the fault coverage against instruction skip is $1 - \frac{1}{2^{10}} = 99.90\%$.

5.4 Security Analysis of Shuffled Pipelined IIR-AES

This countermeasure improves the security of the previous countermeasure by dynamically allocating the positions of the slices within a word. In this work, the slices are rotated by a random number after each encryption. In this scheme, we have 32 different allocations. This reduces the chance of an attacker to inject a useful and undetected fault 32 times because attacker's chance of guessing the position of the target round is 1/32. In addition, this countermeasure significantly reduces the chance of an attacker from repeating the same fault on successive encryptions.

6 Results

In this section we will cover our performance, footprint, and experimental fault coverage. We verified our results in a simulation of a 32 bit SPARC processor called LEON3. We used Aeroflex Gaisler's LEON3 CPU simulator, TSIM. To inject faults and determine the coverage, we wrote a wrapper program [1] for Gaisler's TSIM simulator. The wrapper enabled us to use TSIM commands to inject faults into any instruction, memory location, or register during the execution of the code.

6.1 Performance and Footprint

Our performance and footprint results are presented in Table 2. We wrote a bit-sliced implementation of AES in C and benchmarked it without any fault attack

[1] The wrapper program may be accessed on Github: https://github.com/Secure-Embedded-Systems/tsim-fault

Table 2. Performance and footprint of multilevel countermeasure. Unprotected AES is the reference bit-sliced implementation with no added countermeasure.

	Performance	**Footprint**
Unprotected AES	469.3 cycles/byte	5576 bytes
IIR-AES	1055.9 cycles/byte	6357 bytes
Overhead IIR-AES	2.25	1.14
Pipelined IIR-AES	1942.9 cycles/byte	5688 bytes
Overhead Pipelined IIR-AES	4.14	1.02
Shuffled Pipelined IIR-AES	1957 cycles/byte	6134 bytes
Overhead Shuffled Pipelined IIR-AES	4.17	1.10

Table 3. Experimental fault coverage averages for different fault injections. Every register or instruction in the S-box stage in the last round was targeted one at a time per run for fault injection.

Countermeasure	**Computation Fault Models**				**Instruction Fault Models**
	Random Word	**Random Byte**	**Random Bit**	**Chosen Bit Pair**	**Instruction Skip**
Unprotected AES	0.0%	0.0%	0.0%	0.0%	0.0%
IIR-AES	99.98%	91.45%	93.66%	53.96%	80.56%
Pipelined IIR-AES	100.0%	100.00%	100.0%	98.51%	98.6%
Shuffled Pipelined IIR-AES	100.0%	99.99%	100.0%	98.86%	98.6%

countermeasures added to it. We made three forks of our reference AES implementation and added each stage of our countermeasure to them [2]. Performance was measured by running AES in CTR mode and on a large input size. Footprint was calculated by measuring the compiled program size. Overheads were calculated by dividing by the corresponding reference bit-sliced AES metric.

Using Shuffled Pipelined IIR-AES will be about four times as slow as the reference implementation. Considering it can protect against side channels from the most dangerous of fault attacks, it is a good compromise.

The original AES metric is slow compared to other works because it is an unoptimized implementation. Other works have been able to get bit-sliced AES implementations down to about 20 cycles/byte on 32 bit ARM [14]. We believe the performance overhead for adding IIR would scale with the reference performance.

6.2 Experimental Fault Coverage

We ran fault simulations that emulated our considered adversaries. We injected faults for every register or instruction used in the S-box step of the last round. For data faults, our simulation would enumerate each register, injecting 1 fault, then letting the program run to completion to check the resulting ciphertext. For instruction skips, each instruction is similarly enumerated for skipping and checking the resulting ciphertext.

The S-box step has 144 instructions, consisting of 18 memory operations and 126 computational operations. Of the 144 instructions, 404 operands were registers. Each fault injection simulation was repeated 50 times and averaged together. For

[2] Our implementations can be accessed on Github: https://github.com/Secure-Embedded-Systems/fault-resistant-aes

each data fault simulation, 20,200 faults were injected. For each instruction skip simulation, 7,200 faults were injected.

Table 3 shows the average fault coverages for each countermeasure. Most of the experiments match closely with our theoretical fault coverages. IIR-AES has slightly lower fault coverage then theorized for random bit and byte faults because memory addresses stored in a register can change to a different but valid location, resulting in a control fault. Because of this, the theoretical fault coverage for data faults will be slightly averaged with the control fault coverage. Random bit and byte coverage is slightly lower than expected and chosen bit pair is slightly higher than expected for IIR-AES.

Instruction skip coverage in IIR-AES is 5.56% higher then expected, which is likely just specific to the S-box and key constants we used.

7 Conclusion

We have introduced a set of novel and state of the art methods for detecting faults in block ciphers. We use only software and introduce intra-instruction redundancy. We can protect from well timed, repeatable faults. By adding pipelining, we make our block cipher rounds spatial and much harder to target. And by finally applying random rotations, we make it even more difficult to fault the target round more than once. We show that the performance overhead of our countermeasure is acceptable and scales depending on the desired security level. Our program size overhead is considerably lightweight. We theoretically show why our countermeasure meets the requirements for different fault models. We support our theoretical claims using experimental simulation results based on Gaisler's LEON3 simulator.

8 Acknowledgments

This research was supported in part by the National Science Foundation Grant 1441710, Semiconductor Research Corporation Task 2552.001, and Nation Science Foundation CyberCorps Program.

References

1. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults (extended abstract). In: Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding. (1997) 37–51
2. Ali, S., hyay, D.M., Tunstall, M.: Differential fault analysis of AES: towards reaching its limits. J. Cryptographic Engineering **3** (2013) 73–97
3. Lomné, V., Roche, T., Thillard, A.: On the need of randomness in fault attack countermeasures - application to AES. In: 2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012. (2012) 85–94
4. Barenghi, A., Breveglieri, L., Koren, I., Pelosi, G., Regazzoni, F.: Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In: Proceedings of the 5th Workshop on Embedded Systems Security, WESS 2010, Scottsdale, AZ, USA, October 24, 2010. (2010) 7:1–7:10

5. Guo, X., Karri, R.: Invariance-based concurrent error detection for advanced encryption standard. In: The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012. (2012) 573–578
6. Patranabis, S., Chakraborty, A., Mukhopadhyay, D., Chakrabarti, P.P.: Using state space encoding to counter biased fault attacks on AES countermeasures. IACR Cryptology ePrint Archive **2015** (2015) 806
7. Breier, J., Jap, D., Bhasin, S.: The other side of the coin: Analyzing software encoding schemes against fault injection attacks. In: IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2016, Washington, DC, USA, 2016. (2016)
8. Battistello, A., Giraud, C.: Fault analysis of infective AES computations. In: 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013. (2013) 101–107
9. Battistello, A., Giraud, C.: Lost in translation: Fault analysis of infective security proofs. In: 2015 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2015, Saint Malo, France, September 13, 2015. (2015) 45–53
10. Endo, S., Homma, N., Hayashi, Y.i., Takahashi, J., Fuji, H., Aoki, T.: A multiple-fault injection attack by adaptive timing control under black-box conditions and a countermeasure. In: Constructive Side-Channel Analysis and Secure Design: 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers. Springer International Publishing, Cham (2014) 214–228
11. Ghalaty, N.F., Yuce, B., Taha, M., Schaumont, P.: Differential fault intensity analysis. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on. (2014) 49–58
12. Tunstall, M., Mukhopadhyay, D.: Differential fault analysis of the advanced encryption standard using a single fault. Cryptology ePrint Archive, Report 2009/575 (2009) `http://eprint.iacr.org/`.
13. Guo, X., Mukhopadhyay, D., Karri, R.: Provably secure concurrent error detection against differential fault analysis. Cryptology ePrint Archive, Report 2012/552 (2012) `http://eprint.iacr.org/`.
14. Atasu, K., Breveglieri, L., Macchetti, M.: Efficient aes implementations for arm based platforms. In: Proceedings of the 2004 ACM Symposium on Applied Computing. SAC '04, New York, NY, USA, ACM (2004) 841–845