Software Protection Against Fault and Side Channel Attacks

Conor P. Patrick

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Masters of Science

in

Computer Engineering

June 6, 2017

Blacksburg, Virginia

Keywords: Fault Injection, Side Channel, Countermeasure, Attack

Software Protection Against Fault and Side Channel Attacks

Conor P. Patrick

ABSTRACT

Embedded systems are increasingly ubiquitous. Many of them have security requirements such as smart cards, mobile phones, and internet connected appliances. It can be a challenge to fulfill security requirements due to the constrained nature of embedded devices. This security challenge is worsened by the possibility of implementation attacks. Despite well formulated cryptosystems being used, the underlying hardware can often undermine any security proven on paper. If a secret key is at play, an adversary has a chance of revealing it by simply looking at the power variation. Additionally, an adversary can tamper with an embedded system's environment to get it to skip a security check or generate side channel information.

Any adversary with physical access to an embedded system can conduct such implementation attacks. It is the focus of this work to explore different countermeasures against both side channel and fault attacks. A new countermeasure call Intra-instruction Redundancy, based on bit-slicing, or N-bit SIMD processing, is proposed. Another challenge with implementing countermeasures against implementation attacks, is that they need to be able to be combined. Most proposed side channel countermeasures do not prevent fault injection and vice versa. Combining them is non-trivial as demonstrated with a combined implementation attack.

Software Protection Against Fault and Side Channel Attacks

Conor P. Patrick

GENERAL AUDIENCE ABSTRACT

Consider a mechanical dial lock that must be opened without knowing the correct combination. One technique is to use a stethoscope to closely listen to the internal mechanical sounds and try to pick out any biases in order to figure out the correct combination without having to go through an exhaustive search. This is what a side channel is.

Embedded systems do not have mechanical sound side channels like mechanical locks but they do leak information through power consumption. This is the basis for power analysis attacks on embedded systems. By observing power, secret information from an embedded system can be revealed despite any cryptographic protections implemented. Another side channel is the behavior of the processor when it is physically tampered with, specifically known as a fault attack. It is important that embedded systems are able to detect when they are tampered with and respond accordingly to protect sensitive information.

Side channel and fault attack countermeasures are methods for embedded systems to prevent such attacks. This work presents a new state of the art fault attack countermeasure and a framework for combining the countermeasure with existing side channel countermeasures. It is nontrivial to combine countermeasures as there is a potential for combined attacks which this work shows as well.

Acknowledgments

I want to thank my committee Dr. Patrick Schaumont, Dr. Leyla Nazhandali, and Dr. Gerdes, for mentoring me and overseeing my work. Thank you to everyone in the Secure Embedded Systems Lab for helping me, including Bilgiday Yuce, Nahid Ghalaty, Chinmay Deshpande, and Yuan Yao. Thank you to Dr. Schamont for helping me come up with new ideas and setting a good example for how to write and convey ideas to others. I've learned a lot during my time in grad school.

Also thank you to all of the mentors I've had during undergrad and grad: Bob Lineberry, Jacob Yacovelli, Markus Kusano.

Thank you to my friends and family for supporting me. My mom, dad, and sister; Susan Patrick, Urey Patrick, Meaghan Patrick.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When implementing secure software on desktop computers or servers, there are sufficient resources (computation, RAM, space) to implement a secure system. And adversaries typically won't be able to get physical access to these types of machines. Most computer attacks in these scenarios are remote or aren't specific to any type of computer.

Embedded systems on the other hand, do not have these properties. Embedded devices are much smaller and cheaper. They don't have nearly the computational abilities, RAM, or storage that a conventional computer would have. During NIST's Lightweight Cryptography Workshop in 2015, H. Tschofenig et. al showed that a conventional ARM M3 or M4 chip can take between 100 and 620 miliseconds to compute a P-256 bit ECC signature [50]. This is many times slower than a server computer. Not to mention, ARM is already on the high end of embedded processors. Smaller 16 or 8 bit processors could take another 2-10x as long. This is important to consider because when implementing security measures on an embedded system, it could very well become too costly in terms of time or money.

This challenge is further compounded by the possibility of implementation attacks and an adversary's ability to gain physical access to an embedded system. Embedded systems are becoming ubiquitous. They are in cars, credit cards, door locks, TVs, cable boxes, and other ordinary house hold items that are becoming internet connected. The chance for an adversary to get physical access to something security sensitive is much easier than with conventional computing platforms. With physical access, the adversary can passively tap into the power consumption of an embedded device to conduct power analysis attacks. The adversary can also actively tamper with the processor to try to get it to run into errors or skip instructions, allowing him to bypass a security check or generate a new side channel.

1.1 Power Analysis Attacks

Power analysis started taking hold since the seminal work of P. Kocher et. al [33]. There are different types of analytic techniques. *Simple power analysis* (SPA) is when individual cryptographic operations can be clearly distringuished and attributed to specific key bits. For example, RSA processes key bits individually and determines whether or not substeps of a modular exponentiation is done. SPA will detect the substeps and deduce the outcome of the conditional branches and hence derive the RSA secret key. *Differential power analysis* (DPA) is a step up from that where the adversary makes key guesses and calculates some key-dependent intermediate value for all captured traces and their respective ciphertexts. He then subtracts traces from each other and looks for the largest average difference between the samples of the traces. The max difference should be correlated to the ciphertexts and correct key guesses, allowing the adversary to converge on the correct key. Both techniques require the adversary to have approximate knowledge of where the intermediate value gets processed in the power trace.

A more powerful power analysis technique emerges later, called correlation power analysis (CPA) [14]. It relies on some intermediate value and a power model to correlate the power traces with. For example, for AES, a common intermediate value is the output of the first S-box. A power model would then be some attribute of the intermediate value that is expected to reflect in the power traces; hamming weight is a popular one. So a typical CPA on AES would try to correlate the hamming weight of the first S-box with the power traces. This is done 256 times for all possible key byte guesses and the key guess that provides the best correlation is chosen for the final key guess. An adversary no longer needs to have close knowledge of where the intermediate value is calculated in the power trace, as correlation can be computed against every power sample, only taking the sample with the max correlation in the end (provided there are enough traces to correlate with).

Power analysis techniques continue to improve and grow in statistical complexity. There are also higher order power analysis (HODPA) techniques [36, 30], which will be touched on in the related works section.

1.2 Fault Injection Attacks

Fault injection generically refers to any way that the adversary can meddle with a device's environment to cause a hardware error. A few common methods for how an adversary can do this are given.

Clock glitching: The adversary controls the clock input to a circuit. The adversary can make at least one of the clock pulses shorter, which could violate a critical path of the circuit. Increasing the *intensity* (making the pulse smaller) would likely cause more critical paths to be violated.

Power glitching: The adversary controls the power input of a circuit. He can make a transient where the power changes enough to make internal logic gate voltage signals fall in an undefined range. For example, a 1.5V signal may not clearly evaluate to a logic '1' or '0' in a 3.3V circuit. The intensity of the fault is proportional to the time of the transient and its magnitude.

Laser injection: The adversary has precise control over injecting laser pulses to an exposed die. The radiation induces a current in the circuit or even destroys some part of the circuit.

Electromagnetic injection: A large transient is sent through a coil to try to induce a current in some local part of the target circuit.

Clock glitching and power gliching are common and have the advantage of being relatively easy to set up. Laser injection and electromagnetic injection inject faults local to some area of a circuit and have the potential to make a more precise fault injection. Any individual wishing to learn more about fault injection should consult the Hardware Designer's Guide to Fault Attacks [31].

Using hardware faults to break cryptographic systems was first demonstrated by D. Boneh et. al in 1996 [11]. Differential Fault Analysis (DFA) was soon formalized by E. Biham et. al [10], which breaks generic secret key algorithms. DFA works by considering some intermediate value, say, the output of the last S-BOX, and trying to cause errors in it by injecting faults. One plaintext is encrypted without a fault and that same plaintext is encrypted again with a fault that introduced an error. If the fault causes random n-bit errors, where n is fixed to a small range, to the input of the last S-Box, then a relatively small subset of key bytes could have been possible to create the faulty output. By collecting multiple ciphertext faulty-ciphertext pairs, the intersect of all possible key sets will reveal the correct key byte. The analysis depends on the location of the fault and the severity and repeatability of the error caused by the fault.

After DFA, more efficent fault analysis techniques have been proposed like FSA [34] and DFIA [21] which focus on the magnitude of the errors and the intensity of the fault, rather than the error itself. Well targeted fault injection has been shown to break AES with just a single fault [51].

1.3 Purpose and Outline

In this work, we seek to develop a generic countermeasure that can protect embedded systems from side channel and fault injection attacks. First, we will give an overview of existing countermeasures. Then, we will describe our countermeasure, intra-instruction redundancy, which initially just protects against fault injection. Then, we will describe how our countermeasure can be extended to protect against side channel analysis as well. Being able to combine implementation attack countermeasures is nontrivial, as we demonstrate with an overview of combined DPA and fault injection attacks and our own combined attack analysis.

Chapter 2

Related work

Here we will provide an overview of current fault injection (FI) and side channel analysis (SCA) countermeasures. Some of this is based on related work discussion presented in [57].

B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont. Software Fault Resistance is Futile: Effective Single-Glitch Attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 47–58. IEEE, aug 2016

2.1 Software fault injection countermeasures

Most FI countermeasures rely on some form of redundancy. If there's redundancy, then it is possible to check for errors and form a fault response. Instead of using redundancy, there is the possibility of using sensors to detect for specific changes in the environment [56]. We cover software countermeasures specifically because they are flexible and do not require and modification to the underlying hardware. Thus they could be applied to almost any embedded system. We separate them into algorithm-level and instruction-level countermeasures.

Most countermeasure works do not go over a fault response. I.e., what do you do after you detect a fault? This can be a hard question to answer, and is likely application dependent in most cases. In all cases, however, it must make sure that no output is given to the adversary to prevent him from conducting fault analysis. Ultimately, the countermeasure allows a program to detect faults. What happens next is left to the designer.

2.1.1 Algorithm-level countermeasures

Algorithm countermeasures are those that work at the algorithm level and that do not focus on low-level assembly changes. There are techniques that leverage basic temporal redundancy such as executing an algorithm twice on the same data and asserting the output is the same [6]. This can also be done in encryption by decrypting any output ciphertext and asserting it is the same as the input plaintext [15].

An alternative to temporal redundancy is using information redundancy such as parity bits and error detection codes. Medwed et al. redefine the operations in binary fields to use *AN+B* codes for checking the integrity of data and control flows [35]. Karri et al. propose low-cost, parity-based concurrent error detection techniques for substitution-permutation network block ciphers. Their technique compares the parities of a block cipher's input and output [32].

An honorable mention must be made for infective countermeasures [24] even though all attempts at infective countermeasures have been broken [8]. They work by structuring an algorithm such that if any error is introduced, it will be completely diffused into the output such that it is (ideally) infeasible to do any fault analysis on it. If such a countermeasure worked, it would be nice since it is completely stateless and does not need to make a fault response.

As all of the algorithm-level techniques utilize redundancy, they can be broken by injecting multiple faults such that each redundant copy experiences an equivalent fault. Using careful timing, the adversary aims to inject identical faults in redundant copies of the data or the algorithm. This effectively eliminates the possibility to detect faults. This is done to break algorithm-level countermeasures implemented on RSA [4] and AES [20]. Another serious threat to these countermeasures is skipping the execution of critical instructions by fault injection [9, 44].

2.1.2 Instruction-Level Countermeasures

Instruction-level countermeasures are an improvement over algorithm-level. They will detect or recover from a single fault injection. An adversary will not be able to successfully inject multiple faults throughout the algorithm because any one of those faults will be immediately detected or recovered from. All of these countermeasures make the assumption that it is infeasible to successfully inject consecutive faults that are a few clock cycles apart.

Most of the instruction level countermeasures are redundancy based. There is instruction duplication (ID) and triplication (IT). ID and IT work by executing the same instruction multiple times and checking to see if it produces the same output. Instruction duplication (ID) and instruction triplication (IT), not surprisingly, increase the overhead in performance and footprint by at least two-fold resp. three-fold [7]. ID and IT have been believed to have 100% fault coverage [7].

Another instruction-level method is using a fault-tolerant instruction sequence to prevent an adversary from skipping instructions. In this countermeasure, each instruction is replaced with a secure, functionally equivalent sequence of instructions that are still functional if any one instruction is skipped. This scheme was implemented for ARM architecture by Moro et al. [38], and for x86 architecture by Patranabis et al. [40]. The countermeasure was formally verified by Moro et al. [38].

The ARM instruction replacement scheme was also physically tested by Moro et al. [37]. Using electromagnetic pulses, they demonstrated that the fault-tolerant sequences successfully protect control flow for 32-bit instructions. They also used the instruction replacement scheme for a privilege checking function in FreeRTOS and they were unable to bypass it using a single fault.

Despite these countermeasures being tested and formally verified, they can still be defeated by injecting multiple faults that are back to back. It has been recently shown that a single fault can affect multiple adjacent instructions due to microarchitecture details [57]. The single fault injection model used in previous works cannot be trusted.

2.2 Side channel analysis countermeasures

SCA countermeasures differ in that there is nothing to detect as SCA is passive. So they focus on decoupling key-dependent operations from the power (masking) or reducing the signal-to-noise ratio (hiding). Additionally there are key-rolling methods which rely on changing the key frequently enough such that it isn't possible to collect enough traces to break the key. The current state of the art is called Keymill-LR [47]. State of the art key-rolling requires protocol and hardware level changes which can challenging on most embedded systems, so only masking and hiding techniques will be reviewed.

2.2.1 Hiding

Hiding refers to any method to "hide" the key-dependent power variation amongst non key-dependent power variations (noise), hence reducing the signal to noise ratio.

Random delays Probably the most simple countermeasure is to add random delays, where a delay can just be a few clock cycles. This makes power traces unaligned and unable to be correlated without some preprocessing. A simple delay in the beginning of an algorithm would be simple to defeat as the traces can be statically aligned later. Adding various random delays throughout the algorithm would make it harder, but not impossible. It becomes a matter of the adversary conducting more preprocessing. This countermeasure should not be used by itself to defeat SCA.

Wave dynamic differential logic Wave dynamic differential logic (WDDL) [58] is a technique to make a digital circuit always consume constant power, regardless of the data it operates on. For every logic gate A, there must be a dual, A', with the same inputs. If A and A' are precharged to have the same initial inputs, then any following input combination will always cause the same amount of switching and thus a constant power consumption. It has been shown to be effective in significantly reducing side channel leakage [48]. However, it requires considerable work during chip layout and has been shown to still have some leakages due to manufacturing imbalances and timing differences between A and A' [28].

Voltage regulators Another class of works focus on using internal voltage regulators to decouple the chip power variations from the external power supply variations [55, 27, 46]. Most recently, [17] was able to create a power efficient design. They use a bond wire as the voltage regulator inductor so it would be difficult for an adversary to tamper with. These countermeasures can be challenging to implement as they require IC analog design. They could be bypassed with physical tampering or probing of the die, although not all adversaries may be capable of doing so.

2.2.2 Masking

One of the setbacks with hiding countermeasures is that they are difficult to prove mathematical security for. Many believe that hiding countermeasures can be defeated by taking enough traces to overcome a relatively small signal to noise ratio.

Masking countermeasures shine over these setbacks as they are provably secure and taking more traces will not help break a properly masked implementation with normal CPA. They work at a higher level and use random numbers to decouple key-dependent operations from power variation.

Boolean masking Boolean masking works by generating a random number, R, and XOR'ing it with the input plaintext P to create masked plaintext M. Each linear operation $L()$ is applied to M and the random number R. Let C be equal to the unmasked intermediate variable calculated by $L(P)$. We have the following.

$$M = P \oplus R$$

$$M := L(M, key)$$

$$R := L(R, key)$$

$$C := M \oplus R$$

Regular CPA won't be able to use the calculation C to get correlation since R prevents a power model from being created. This can be thought of splitting a secret variable C into 2 separate random shares, C_0 and C_1, where $C = C_0 \oplus C_1$. This isn't a complete computation, as it needs to be able to handle non-linear operations as well. As is formulated by Ishai et. al [29], a non-linear masked AND can be performed using any amount of shares. It needs to have $(d + 1)$ additional random bits per AND, where d is the number of shares. Later, E. Trichina showed how to do it using only one additional bit for a 2-share implementation [49].

Consider an AND gate, with two inputs A and B, each of which are split into two secret shares. Let R be a single random bit. Let C_0 and C_1 be the output shares.

$$C_0 = R$$

$$C_1 = ((((A_0B_0 \oplus R) \oplus A_0B_1) \oplus A_1B_0) \oplus A_1B_1$$

It is important to consider using more than two secret shares due to the possibility of higher order DPA [36]. An adversary can do preprocessing on collected power traces to add together the respective portions of the power trace that different shares are calculated under. Then the samples are squared so the sign is always the same. The goal, is by combining the power samples of the separated shares, to cancel out the effect of the random mask and get a trace that is correlated to the key-dependent variable. Nth order analysis would be used to try to break (N-1) order masking. For HO-DPA attacks to work, close knowledge of where different shares are computed is needed, and likely an additional magnitude of traces will need to be collected to overcome the compounded noise. There are many statistical techniques that can be employed for higher order analysis [23]. Generally, higher order attacks are much more difficult to perform.

It is also possible to use multiplicative masking instead of boolean masking. For example, in AES, the S-BOX can be multiplied by random numbers with respect to $GF(256)$ [25]. This tends to be more computationally intense. The latest state of the art masking AES implementation for ARM using boolean masking and requires 5,248 random bits per 128-AES encryption and runs at 58 cycles/byte on ARM M4 [45].

Chapter 3

Intra-instruction redundancy

This chapter is based on work previously presented in [42].

C. Patrick, B. Yuce, N. F. Ghalaty, and P. Schaumont. Lightweight fault attack resistance in software using intra-instruction redundancy. Cryptology ePrint Archive, Report 2016/850, 2016. `http://eprint.iacr.org/2016/850`

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.

3.1 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.

3.2 Proposed Software Countermeasures for Fault Attacks

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 present spatially in all instructions without actually having to re-execute anything. Because the redundancy is interleaved with the data in every instruction, 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 [19] and microarchitecture effects [57].

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

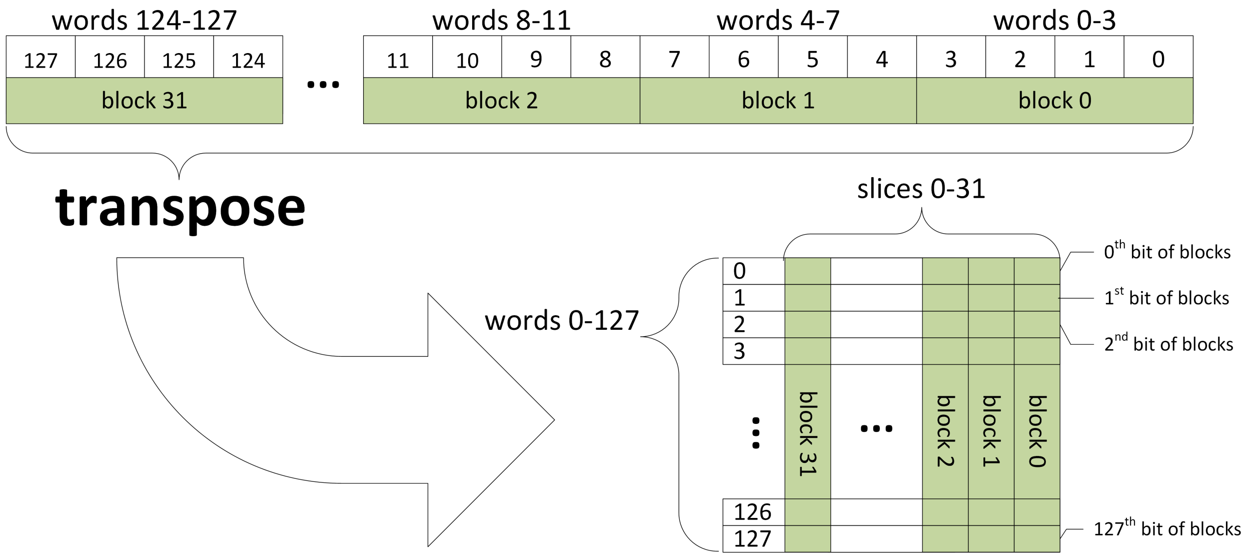

Figure 3.1: Transpose of 32 blocks to fit bitwise into 128 32-bit words.

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.

3.2.1 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 is also useful for timing side channel resistance because it is less timing dependent. 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.

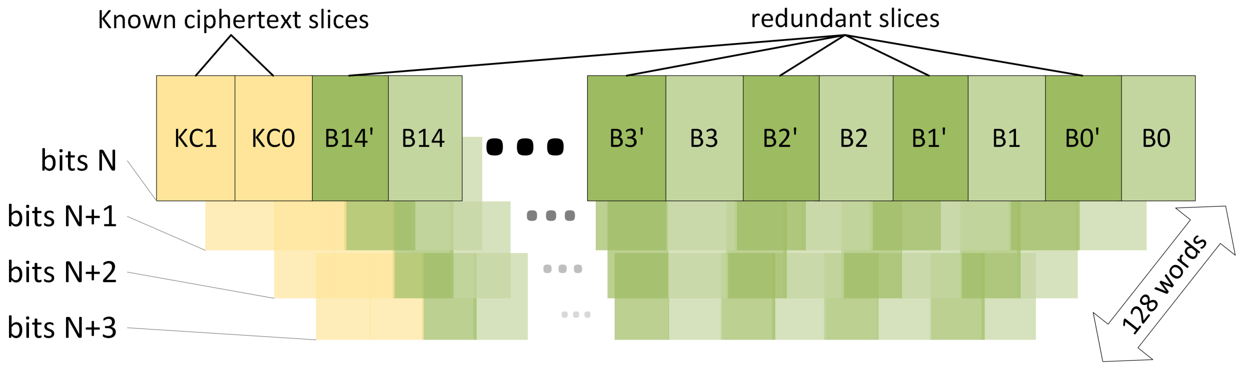

Figure 3.2: Bit-slicing with Intra-Instruction Redundancy using 15 data (B), 15 redundant (B'), and 2 known ciphertext (KC) slices. Each KC slice is aligned with its corresponding round key slices in other words.

A prerequisite for bit-slicing is to transpose the layout of input blocks, as shown in Figure 3.1. 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.

3.2.2 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 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

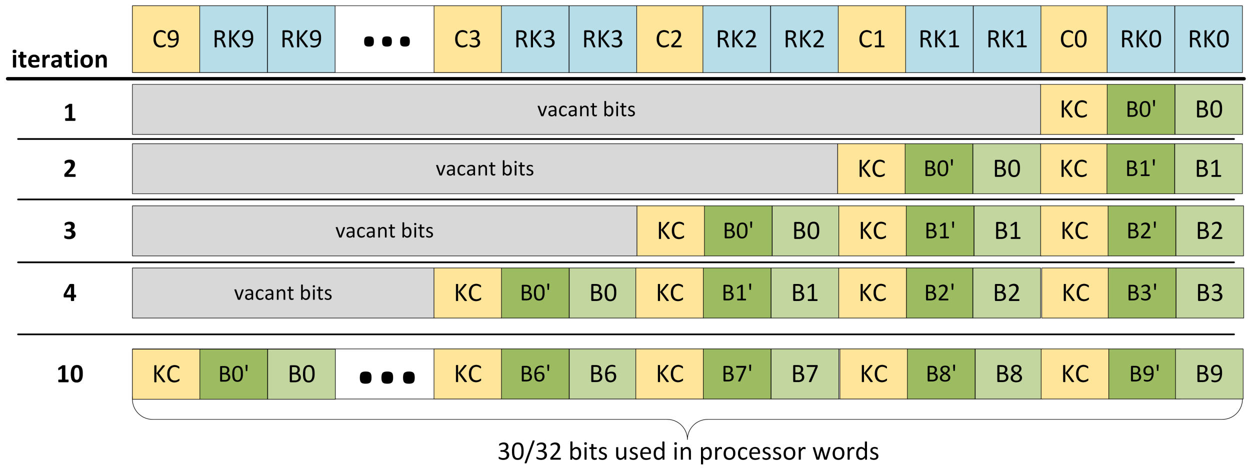

Figure 3.3: Pipelined bit-sliced layout for 32 bit processor. $RK0-9$ are ten different round keys. $B0-9$ are different input blocks and $B'09$ are their redundant copies. KC is a known ciphertext slice. $C0-9$ are the round keys used to produce the known ciphertext.

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.

3.2.3 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 [22, 52]. 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 3.3, 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.

3.2.4 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.

3.2.5 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.

3.3 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

Table 3.1: 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

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.

3.3.1 Performance and Footprint

Our performance and footprint results are presented in Table 3.1. We wrote a bit-sliced implementation of AES in C and benchmarked it without any fault attack countermeasures added to it. We made three forks of our reference AES implementation and added each stage

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

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 [3]. We believe the performance overhead for adding IIR would scale with the reference performance.

3.3.2 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 computa-

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

Table 3.2: 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%

tional operations. Of the 144 instructions, 404 operands were registers. Each fault injection simulation was repeated 50 times and averaged together. For each data fault simulation, 20,200 faults were injected. For each instruction skip simulation, 7,200 faults were injected.

Table 3.2 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.

Chapter 4

Adding side channel protection

If an adversary has physical access to an embedded system and it is his goal to reveal secret information hidden inside, he isn't limited to just SCA or just FI. The adversary is likely to take the easiest path available to reveal wanted secrets. Thus, a designer really needs to provide countermeasures against all types of attacks to be safe from physical implementation attacks.

While IIR can provide good fault resistance, it doesn't help against SCA. On the contrary, it may even make power analysis easier because key-dependent variables have been duplicated which may amplify the signal to noise ratio. So a designer shouldn't use IIR by itself in the general case as the block cipher would still be vulnerable to SCA. This is why we look into adding SCA protection to IIR.

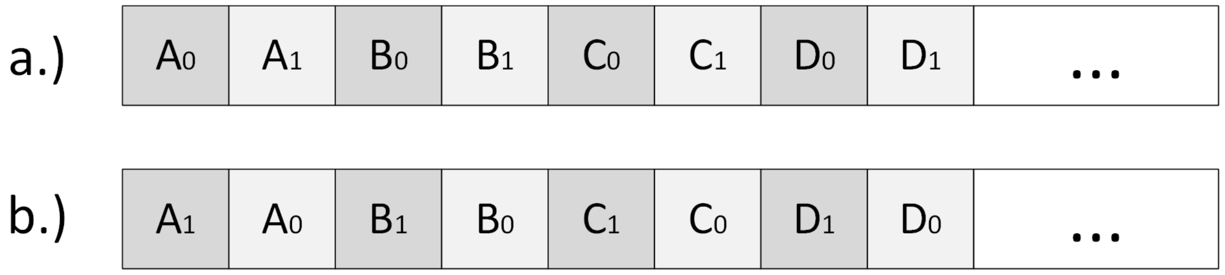

Figure 4.1: a.) Private-slices bit-slice allocation for order-1 protection. A-D are 1 bit data streams that are each represented by two shares. b.) Private-slices word after the custom rotation operation is applied.

4.1 Private slices

Inspired by Ishai's work on private circuits, we propose the use of private slices, which is a formulation of secret shares in a bit-sliced format, as shown in Figure 4.1.a. This could work on a processor with SIMD (same instruction multiple data) extensions, or on a regular processor with bit-slicing techniques. We will assume a regular processor is being used.

For linear operations, everything works the same as with a regular bit-sliced implementation. But for masked AND operations, recall that we need to calculate A_0B_1 and A_1B_0 which require breaking the bit-slice format as the two variable are in different slice locations. We are able to get around this by using a lookup table to implement a custom rotation operation, as shown in Figure 4.1.b. The rotation allows aligns the variables in a different bit-slice word so a masked AND can continue to be computed in bit-sliced fashion.

The code in Figure 4.2 shows how the masked AND can be implemented in C code. Note this is for a 32-bit processor and the random two bytes that are generated are formatted to fit in every other bit of a 32-bit word.

4.2 Results

To evaluate the performance and the side-channel characteristics of Private Slices, we built a prototype bit-sliced AES design in AES128-CTR mode. Our AES implementation combined MixColumns and ShiftRows into one step and used a computational depth-16 S-BOX implementation of depth 16 by Boyar and Peralta [13]. Using this design as a basis, we then constructed five different versions as listed in Table 4.1.

For the masked designs, we assume the existence of a good-quality hardware random number generator. Such a random number generator is a platform-specific design, and we did not include any overhead, nor any random-number distribution imperfections in our evaluations.

We measured side-channel leakage from a 50MHz LEON3 core, configured in the main FPGA of a SAKURA-G board. We used OpenADC [39] which is capable of 105 Msamples/second. We configured it for low gain to achieve the best signal quality. Power traces for the different countermeasures versions can be found in Appendix 1.A. We also tested our code on three different platforms (ARMv7, LEON3 and Intel i7) and collected performance data.

Over the following subsections, we discuss the results from side-channel analysis, performance analysis as well as additional insights obtained from this experimental evaluation.

Table 4.1: We evaluated 5 different versions of AES128-CTR. Three of them (PS and PPS) are proposed by this thesis, the other two (BS, BM) are for comparison.

Countermeasure	**Description**
BS	Unmasked bit-sliced AES128-CTR.
BM	A first-order masked bit-sliced AES128-CTR. This design uses separate processor registers for separate shares, and it is similar to related work [5, 45].
PS	A first-order Private Slices AES128-CTR.
PPS	A first-order Pipelined Private Slices AES128-CTR.

4.2.1 SCA Resistance

To make sure our implementations were vulnerable to SCA on our platform, we conducted a DPA attack on our unprotected bit-sliced AES design (BS), from which all of our countermeasures were forked from[1]. We found we could recover the full key of the unprotected AES implementation with about 300-500 traces. Our attacks aim at the first round, using the first AddRoundKey and S-Box as the leakage model.

We performed Welch t-tests [26] to evaluate the side channel leakage of our countermeasures. The Welch's t-test can determine how well two different datasets can be distinguished from each other. In our case, we took 12K measurement traces of each countermeasure during

[1]Our implementations will be released as open source.

Table 4.2: First order Welch t-test for first round output bits. Values between -4.5 and 4.5 are considered to be side channel leakage secure with 99.999% confidence. Max and Min are over the 128 output bits of the first round. Mins, Maxes, and Averages are conducted on the absolute values.

Countermeasure	**Max Leakage**	**Bit #**	**Min Leakage**	**Bit #**	**Average**
BS	-87.422	99	-14.531	126	40.500
BM	4.139	106	1.595	81	2.648
PS	3.305	67	1.798	78	2.446
PPS	4.202	72	-1.632	91	2.573

the first AddRoundKey and S-Box. The plaintext was random and the key was fixed. The traces were then filtered into two groups: group A, where state bit N was 1 after add round key and S-Box, and group B, where state bit N was 0 after add-round-key and S-Box. The t-test was then computed for all group pairs for all 128 state bits. This determines if group A can at all be distinguished from group B for all samples in a trace. The maximum statistic magnitude for each bit was taken. If the result for each bit is between -4.5 and 4.5, then there is no leakage with a confidence of 99.999%. All tests were repeated multiple times with different datasets to reduce noise and ensure our results are consistent.

First-Order Leakage In Table 4.2, the first-order leakage is shown. Unprotected AES, not surprisingly, has the most leakage. The masking implementation passes as expected, albeit some careful work to prevent the compiler from causing an accidental unmasking effect.

Table 4.3: Second order Welch t-test for first round output bits. The same traces and calculations are done as in Table 4.2 except preprocessing is done to make it suitable for second order analysis.

Countermeasure	**Max Leakage**	**Bit #**	**Min Leakage**	**Bit #**	**Average**
PPS	-5.69	83	1.72	62	2.85

The compiled induced unmasking effect does not occur in Private Slices. The Private Slices version (**PS**), as well as the Pipelined Private Slices (**PPS**) passes the 4.5/-4.5 condition. Figure 4.3 shows the various t-statistics across a subset of trace samples. It shows that standard unmasked bit-sliced design (**BS**) cross the 4.5/-4.5 boundary multiple times, while masked (BM), private and pipelined-private do not.

Second-order Leakage In Table 4.3, the second-order leakage is shown for the Pipelined Private Slices version (**PPS**). The test was done on the same traces after the zero-offset preprocessing [53]. Pipelined Private Slices had one bit that failed with a t-statistic of -5.69, along with others failing the 4.5/-4.5 bound. This was during the first round, so only one mask and mask data slice was being processed at a time. We plan to investigate the hiding effect of the Round Pipelining as future work.

The Danger of Regular Masking When we first implemented a regular bit-sliced masking scheme in C code (**BM**), we found that it kept getting residual side channel leakage. This is because the compiler will overwrite registers, that previously held a share, with the other share. This share-on-share write causes leakage directly linked to the original data. We wrote a simple assembler stage that zeros out registers in-between instructions that would cause share-on-share writes. Figure 4.4 shows the t-test statistics for the pure C implementation of masking and the same implementation with the assembler stage fix. It is clear the accidental leakage is significant. **PS** avoids this because shares are never aligned between registers except in the isolated masked AND operation.

4.2.2 Performance

Lastly, we evaluated the performance of our countermeasures. We tested on three different platforms: ARMv7, LEON3, and Intel i7. Our implementations are programmed in C, so the same implementations are tested on each platform. We measured cycles/byte counts by using a large enough plaintext input that would maximize the bit-sliced throughput. On a 32-bit implementation for AES-128, that is 512 bytes. We did not include the key schedule computation in our measurements.

The performance and program size (footprint) results are shown in Table 4.4 and they are ranked by cycles/byte. All designs are optimized for performance. The performance scales with the complexity and security level of each countermeasure for each platform, as expected. For **PS** and **PPS** countermeasures, the majority ($> 90\%$) of the computation overhead is spent in the S-box, mostly for the masked AND operation. Hence, the performance improvement of a custom instruction for the share-rotate operation would be significant.

Table 4.4: Throughput and program size metrics for all countermeasures on various platforms. Units are in cycles/byte (c/b) and bytes (b). The same implementation of each countermeasure is tested on each platform.

	LEON3 Spartan6 50MHz		**Intel i7 4GHz**		**ARMv7 850 MHz**	
Countermeasure	**Performance** (cyles/byte)	**Footprint** (byte)	**Performance** (cyles/byte)	**Footprint** (byte)	**Performance** (cyles/byte)	**Footprint** (byte)
BS	398.0	9,280	80	25,544	181.04	11,424
Overhead	1	1	1	1	1	1
BM	877.2	14,516	130	-	457.79	-
Overhead	2.20	1.56	1.63	-	2.53	-
PS	1,565.3	13,588	315	-	735.71	-
Overhead	3.93	1.46	3.94	-	4.06	-
PPS	2,714.5	10,884	795	-	1,154.40	-
Overhead	6.82	1.17	9.94	-	6.38	-

Footprint overhead is shown only for LEON3, as its compiler does not do advanced loop unrolling and cache-sensitive optimizations. This gives a better estimate for constrained devices.

On ARMv7, our Private Slices performance (4x) overhead improves upon previous masking implementation overhead (5x) by Balasch et. al. [5]. Their implementation requires 6,048 random bits per encryption while ours requires 5,568. This includes 128 bits per block for the initial mask and 544 bits per S-box (34 AND gates) operation per AES round. Schwabe and Stoffelen in [45] implement a highly optimized masked AES implementation on ARM M4. Their code size overhead is 3.29 and their throughput overhead is 3.27 (not including random number generation). Since our implementation relies on an optimized Secure AND construction and not optimized C code, this is an acceptable difference. Their implementation uses 5,248 bits per block as they had 2 less AND operations in their S-BOX than ours.

4.2.3 Further Analysis

The throughput overhead is mostly due to the masked AND operations in the S-box. When removing the masked AND operation on the ARM platform for 3rd order PS, there is about 10x less overhead. A first-order masked AND only needs to do one bit-swap operation, while a third order needs to do three. Since the bit-swap is unconventional for typical instruction sets, it needs to be implemented in software. We choose to do it with a byte-wise lookup table, meaning a 32-bit word needs to complete 4 memory loads and 6 shifting operations to complete one bit-swap. This could be greatly improved as the swap bits operation can be implemented with just wires in hardware. A second reason for using the lookup table is that the generation of side-channel leakage by overwriting shares between registers is minimized. We could have used assembly programming to avoid this, but this results in a non-portable solution. Private Slices therefore is better suited for implementation in a high-level language. Additionally, we believe Private Slices will scale better with program size, when higher-order masking is required. Using plain masking (**BM**), going to higher-order means introducing additional variables for each share. In Private Slices, going to higher order means allocating more bit-slices per bit and reducing the throughput.

4.3 Future Work

Private slices shows how private circuits can work in software, in a bit-sliced fashion like IIR. We've shown that the implementation continues to be side channel resistant and has comparable performance metrics to state of the art masking implementations. Note that our metrics aren't necessarily better, as that wasn't the intended goal. We are trying to formulate a combined countermeasure.

Private slices and IIR hasn't been combined yet but we suspect it will be successful. It would be a matter of adding the masked AND operation and the fault redundant slice management, both of which are relatively small pieces of code. An example of how a bit-slice allocation could be made is given in Figure 4.5.

Ultimate goal The target application doesn't have to be a block cipher but rather could be any application. Any boolean operation implemented with private slices and IIR so any hardware circuit can be virtualised in our software countermeasure. The performance may not be good, but it may have the advantage of making any arbitrary program fault injection resistant and side channel resistant on any platform. Our work could be made into a primitive instantiated as virtualised software or hardware.

```
static  word_t SAND(word_t p, word_t q)

{

        word_t r1 = random_2bytes();

        word_t n1 = p & q;


        word_t qswap = bitswap[(uint8_t)q]

        |  ((word_t)bitswap[(uint8_t)(q>>8)] << 8)

        |  ((word_t)bitswap[(uint8_t)(q>>16)] << 16)

        |  ((word_t)bitswap[(uint8_t)(q>>24)] << 24);


        word_t n3 = p & qswap;

        word_t n4 = r1 ^n1;

        word_t z = n3 ^n4;


        return z;

}
```

Figure 4.2: C code implementation of bit-sliced masked AND.

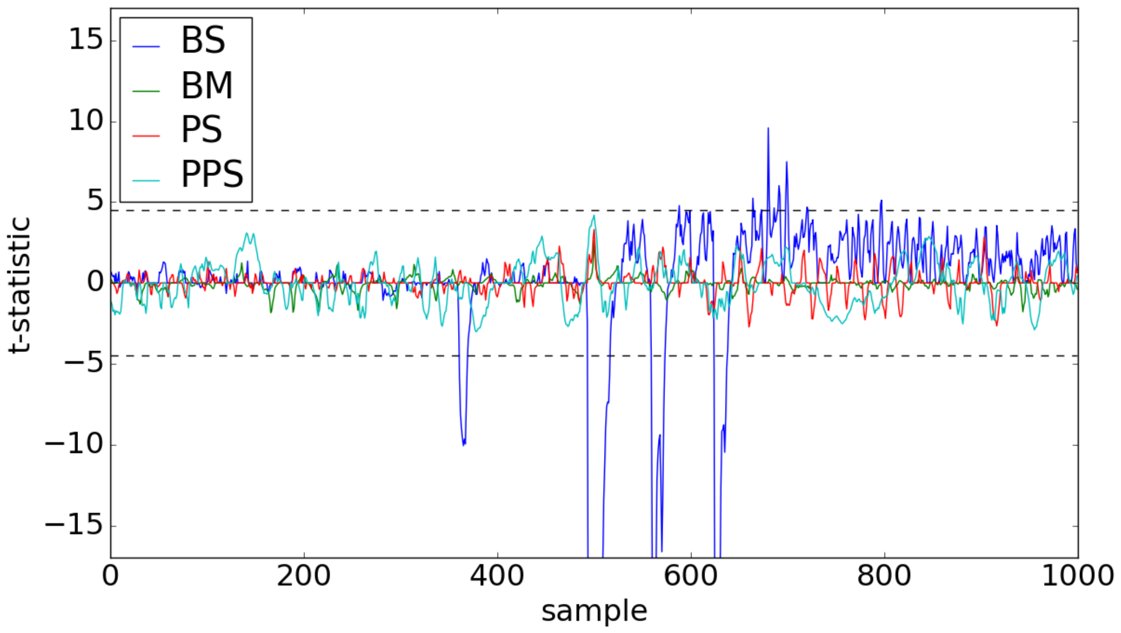

Figure 4.3: Overlaid plots of the t-statistics for each countermeasure, each centered around the maximum t-values. Regular bit-sliced (BS) fails the leakage test. Masked (BM), Private Slices (PS), and Pipelined PS (PPS) pass as they stay between the boundaries.

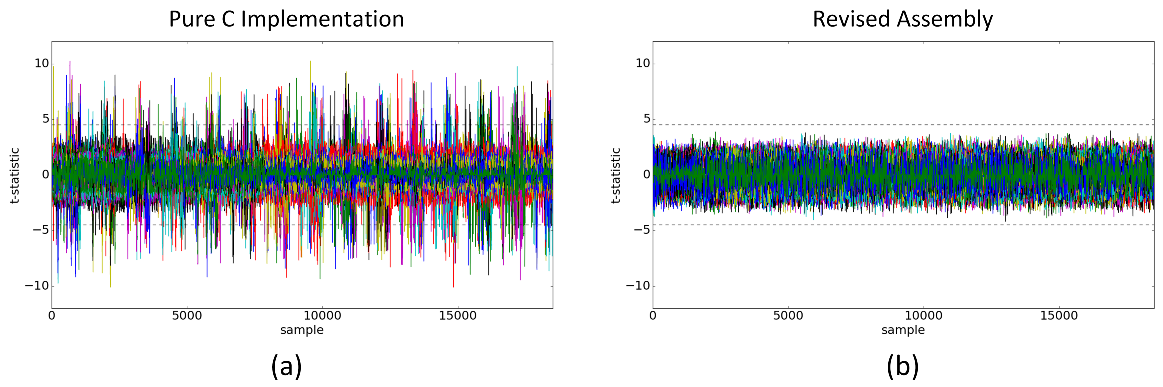

Figure 4.4: Welch t-statistics for all 128 output bit tests. (a) Masking implementation purely in C allows compiler to cause accidental leakage. (b) The same implementation with an assembly pass to fix compiler placed share-to-share register writes.

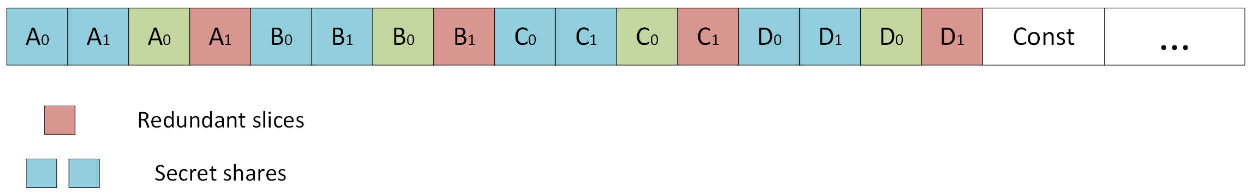

Figure 4.5: Example bit-slice allocation for combining IIR and private slices.

Chapter 5

Combined attacks

This chapter is based on work submitted to eprint [41].

C. Patrick, Y. Yao, and P. Schaumont. Addroundkey power model for fault injection enabled cpa attacks. Cryptology ePrint Archive, Report 2017/xxx, 2017. `http://eprint.iacr.org/2017/xxx`

We revisit the idea of combined fault injection and side channel analysis attacks (combined attacks). More specifically, we run some experiments to test the feasibility of fault-enabled SCA on masked implementations. As stated previously, perhaps the most popular SCA countermeasure is masking. Nearly all masking implementations require a random mask to be added at the start, right before AddRoundKey. We target faulting the mask addition here to enable SCA on AddRoundKey. This type of attack could be applied to any masking implementation, even with the presence of fault countermeasures. We show that the fault model is relatively simple and repeatable, as well as using AddRoundKey as a power model.

No previous combined attack, to the extent of our knowledge, has actually been demonstrated on an masked, fault resistant implementation. We plan to explore the feasibility of a combined attack that should be applicable to any masked block cipher implementation. It should also be able to bypass most fault resistant block cipher designs. A typical masked and fault tolerant block cipher implementation is potentially still vulnerable to combined attacks.

The reason for this chapter is to demonstrate that combined attacks could be practical and that the composability of FI and SCA countermeasures is not great. In other words, it is still difficult to effectively combine FI and SCA countermeasures. In a typical threat model, an adversary has physical access to an embedded device and desires to leak sensitive information from it. If the device only has an SCA countermeasure, and no FI countermeasure, it would likely suit the adversary to use DFA or some other fault injection attack. Thus for most designers to ensure implementation attack security, they must include composable countermeasures for both FI and SCA. As we show in this chapter, composability is not a given – combined countermeasures must be tested against SCA and FI combined, not separately.

5.1 Previous combined attack works

Combined fault and side channel analysis work starts in 2006 with F. Amiel et. al's work on fault analysis of DPA resistant algorithms [1]. They give a number of techniques to use faults to attack the initialization of a masked AES smart card implementation. Attacking the initial masked AddRoundKey with faults skips the loop early so that only one of the key bytes is used with one byte of random mask R. A search is done to find R by changing the input plaintext byte that corresponds to R until a colliding value is found. Then R is known and the first key byte can be found through brute force. This is repeated with all key bytes. A. Boscher and H. Handschuh later improve this work [12] to show that fault injection attacks can work regardless of any masking scheme. None of these attacks would not work on a fault tolerant system as it would not have access to ciphertext in the advent of a successful fault.

Soon after F. Amiel et. al's initial work, they introduce Passive and Active Combined Attacks (PACA) [2]. They demonstrate they can fault the initialization of a random mask in RSA and use simple power analysis (SPA) to reveal the key. They make the point that even if RSA detects the fault and doesn't output ciphertext, it would be too late as it already did the computation before finding the fault. C. Clavier et. al extend this further to work on masked AES [16]. They use a similar collision search to figure out what the error differential was introduced by the fault and discover the value of $K \oplus R$. They then show they can do correlation using key guesses and random number guesses; effectively a 16-bit correlation search rather than an 8-bit correlation search with regular CPA. It is a clever attack but it requires a lot of computation and does not work if the target implementation has fault resistance.

T. Roche et. al [43] in 2011 propose a combined attack that can defeat a masked block cipher that is DFA resistant. So they do not need the output ciphertexts and only have control over the input. They require two faults to be injected; each on the last two round keys during the key schedule. They then make a 24-bit guess encompassing the key byte guess and error bytes from each fault and do correlation on the faulty power traces. This approach may not always be feasible because it requires the attacker to be able to inject a fault during key schedule and synchronize it with a fault in a later round. The computation required for correlation over 24-bits is relatively high as well. The required fault model is simplified in 2012 [18] but the key schedule fault injection is still required. Both Roche's and Dassance's works are theoretical in that no experiment is performed.

To the extent of our knowledge, we don't know of any other relevant combined attacks works on block ciphers since 2012. We propose a method that may be more practical. The fault injection just needs to disable the application of the mask to one of the bytes at the start of encryption and then CPA can be done on the AddRoundKey step. The practicality of this is demonstrated.

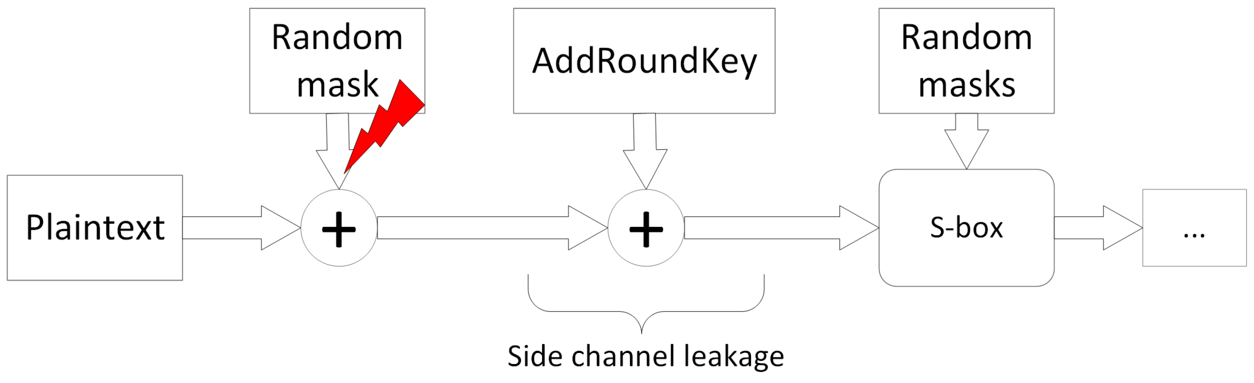

Figure 5.1: Attack model for proposed combined attack. The initial random mask is disabled using a fault which allows side channel analysis to be conducted on the AddRoundKey step of AES.

5.2 Attack Model

We assume that our target is AES, uses boolean masked, and has fault resistance such that it will detect a fault at the end of encryption [6, 15, 32] or during encryption [7, 38]. Thus, an adversary can never see faulty ciphertext but can observe the power variation of the faulty ciphertext calculation. Latest first order boolean masking implementations require a 16 byte random mask [45] at the start per encryption and 640 more bytes throughout the encryption to cover the S-box computations, assuming one block being encrypted per encryption.

The target system is given plaintext input as controlled by the adversary. The generated random mask must be applied to the plaintext at some point through XOR for the boolean masking implementation to work. We assume that the adversary can successfully skip one of these instructions, effectively leaving at least one plaintext byte unmasked as it goes through the AES AddRoundKey step. The AES encryption should continue after a successful fault as no faulty ciphertext is created. Thus $P_n \oplus K_n$ is potentially leaking key information, where P_n is unmasked plaintext byte n and K_n is key byte n. Regular CPA can be conducted to reveal K_n using $P_n \oplus K_n$ as the target key-dependent intermediate value. This model is depicted in Figure 5.1.

We address a few apparent challenges with this approach.

How to successfully skip the mask XOR? An adversary first has to locate the start of encryption which can be done via via simple power analysis. Knowing the implementation uses masking, the adversary can initially keep the input plaintext constant. By looking at the variation across of many power traces, a few potential locations of the mask initialization can be located. This is because the variation will be relatively high when random numbers are handled even when the input is constant. Then the adversary has a short window of time to enumerate for fault injection. To test if a particular fault injection was successful in skipping a mask initialization, the adversary should make two tests. First, is the ciphertext correct? Second, does the constant-plaintext power variation decrease? If yes to both, the fault injection was likely successful. If a random byte was disabled, then the constant-plaintext power variation would be expected to go down. For higher-order masked implementations, there may need to be multiple fault injections, or a better targeted single injection [57].

Why not try skipping masks in S-box? While this approach is possible, it may not be feasible. Consider initial mask M and first S-box mask N. This attack requires the adversary to not only skip the mask M_n but then inject a second fault for S-box N_n where M_n and N_n correspond to the same byte n. The attacker doesn't necessarily know the location of where N_n is used and the constant-plaintext variation test may not help to test if the fault was successful since it could provide false positives.

Is AddRoundKey leaky enough or not? AddRoundKey step of AES is just one XOR operation for a particular key byte. Compared to the popular S-Box CPA target, AddRoundKey is much smaller. It is also a linear operation, so there are going to be "ghost" keys that will generate the same correlation as the correct key, as shown in the results section. It may be easier to just perform a higher-order DPA; There could be a lot of factors that determine what attack strategy is successful. We show that a combined attack could break a masked AES application on a lightweight RISC-V FPGA implementation [54]. We compare it to using the normal Hamming weight S-box model.

5.3 Results of AddRoundKey power model

Conducting CPA with a power model that is based on a linear operation like AddRoundKey will have the unwanted effect of ghost keys. This is because for any particular key byte guess K that has correlation s for all traces, its opposite $\sim K$ will have exactly $-s$ correlation.

Table 5.1: The number of traces needed to reveal an AES key on RISC-V processor for S-box and AddRoundKey power models. * Ignores the ghost keys. *Note this table is currently incomplete and will be filled in later.*

Power model	Key bytes broken	Number of traces
S-box	16	1000
AddRoundKey	16*	10000

This is because the Hamming weight of AddRoundKey p^K, $HW(p^K)$, is linearly related to $\sim K$ by equation $HW(p^K) = 8 - HW(p^{\sim}K)$. So each key byte entropy can at best be reduced to 1 bit, or 16 bits total for an AES-128 key.

Table 5.1 shows the results of a CPA attack on a AES implementation that runs on RISC-V on an FPGA using C. Wolf's lightweight RISC-V implementation [54]. It takes about ten times more traces to break AES using an AddRoundKey Power model than it would using an S-Box model on an unmasked implementation. Also the confidence margin for AddRoundKey is much smaller. The table ignores the effect of the ghost keys, but really it leaves 16 bits of entropy to brute force, which is trivial.

5.4 Potential improvements for combined countermeasures

One of the challenges with an attack like this is that it doesn't aim to produce faulty ciphertext. However, during setup and fault discovery for this attack, faulty ciphertext would likely be triggered. So one countermeasure could be to have a destructive fault response, i.e. self-destruct, betting the adversary would trigger this accidentally. But this could be presumptuous for some applications. An application could further protect itself by checking the integrity of the generated mask or use a key rolling technique.

Additional Integrity Checking . The random number generation could provide some means of redundancy such that the integrity of the mask can be checked after it has been applied to the plaintext. If the same random number is generated twice, a redundant check could be made. Although a designer should still be careful as fault injection can bypass multiple checks with a single fault [57]. A design that can incorporate spatial redundancy would be appropriate [42].

Key rolling . If the core source of entropy (the key) can be secured from implementation attacks, combined attacks should no longer be practical. Keymill-LR [47] was recently presented at HOST and shows promise as a side channel resistant key rolling technique. It is not a complete solution by itself as it does not has fault resistance. If it incorporates redundancy in such a way it is non-trivial to bypass with a fault, it could be a potential combined countermeasure. It could require some more work to implement as key-rolling requires some protocol level changes.

We show a potential practical method to defeat a "canonical" combined countermeasure. The fault model relies solely on being able to skip an instruction in the beginning and it is possible to test if it was successful without using faulty ciphertext. It relies on the

AddRoundKey power model and we've shown it is viable to use, albeit requiring many more traces than with a S-box model. It may be a good alternative to performing a higher-order DPA attack. We believe that this combined attack is practical and that there needs to be more work on combined countermeasures.

Chapter 6

Conclusion

With this thesis we introduce a method to counter fault injection attacks in software. Hardware solutions are slow to market which leaves designers to try to develop countermeasures in software. Many conventional countermeasures are based on temporal redundancy and can be broken, which sets the stage for intra-instruction redundancy which is based solely on spatial redundancy despite being implemented in software. We've shown IIR can be extended to protect against side channel analysis using our implementation of masking and can be safely implemented in a higher level language without the compiler causing accidental leakage. It is important for an implementation attack resistant design to be protected from both SCA and FI as any realistic adversary would be capable of both. We provide a review of combined attacks as they are applicable to combined countermeasures and present the practicality of doing a combined attack that uses AddRoundKey as a power model. We conclude that a combination of spatial redundancy and integrity checking at the entropy source or key rolling as a potential countermeasure to end SCA and FI based attacks.

Bibliography

[1] F. Amiel, C. Clavier, and M. Tunstall. Fault Analysis of DPA-Resistant Algorithms. pages 223–236. Springer, Berlin, Heidelberg, 2006.

[2] F. Amiel, K. Villegas, B. Feix, and L. Marcel. Passive and Active Combined Attacks: Combining Fault Attacks and Side Channel Analysis. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, pages 92–102. IEEE, sep 2007.

[3] K. Atasu, L. Breveglieri, and M. Macchetti. Efficient aes implementations for arm based platforms. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 841–845, New York, NY, USA, 2004. ACM.

[4] C. . Aumüller, P. . Bier, P. Hofreiter, W. Fischer, and J.-P. Seifert. Fault attacks on RSA with CRT: Concrete Results and Practical Countermeasures. Cryptology ePrint Archive, Report 2002/073, 2002. `http://eprint.iacr.org/`.

[5] J. Balasch, B. Gierlichs, O. Reparaz, and I. 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.

[6] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[7] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni. Countermeasures Against Fault Attacks on Software Implemented AES: Effectiveness and Cost. In *Proceedings of the 5th Workshop on Embedded Systems Security*, WESS '10, pages 7:1–7:10, New York, NY, USA, 2010. ACM.

[8] A. Battistello and C. Giraud. Fault Analysis of Infective AES Computations. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 101–107. IEEE, aug 2013.

[9] A. Battistello and C. Giraud. Fault cryptanalysis of ches 2014 symmetric infective countermeasure. Cryptology ePrint Archive, Report 2015/500, 2015. `http://eprint.iacr.org/`.

[10] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. pages 513–525. Springer, Berlin, Heidelberg, 1997.

[11] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. pages 37–51. Springer, Berlin, Heidelberg, 1997.

[12] A. Boscher and H. Handschuh. Masking does not protect against differential fault

attacks. In *Fault Diagnosis and Tolerance in Cryptography - Proceedings of the 5th International Workshop, FDTC 2008*, pages 35–40. IEEE, aug 2008.

[13] J. Boyar and R. Peralta. A depth-16 circuit for the aes s-box. Cryptology ePrint Archive, Report 2011/332, 2011. `http://eprint.iacr.org/2011/332`.

[14] E. Brier, C. Clavier, and F. Olivier. Correlation Power Analysis with a Leakage Model. pages 16–29. Springer, Berlin, Heidelberg, 2004.

[15] M. Ciet and M. Joye. Practical Fault Countermeasures for Chinese Remaindering Based RSA (Extended Abstract). In *IN Proc. FDTC05*, pages 124–131, 2005.

[16] C. Clavier, B. Feix, G. Gagnerot, and M. Roussellet. Passive and Active Combined Attacks on AES Combining Fault Attacks and Side Channel Analysis. In *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 10–19. IEEE, aug 2010.

[17] D. Das, S. Maity, S. B. Nasir, S. Ghosh, A. Raychowdhury, and S. Sen. High efficiency power side-channel attack immunity using noise injection in attenuated signature domain. *CoRR*, abs/1703.10328, 2017.

[18] F. Dassance and A. Venelli. Combined Fault and Side-Channel Attacks on the AES Key Schedule. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 63–71. IEEE, sep 2012.

[19] S. Endo, N. Homma, Y.-i. Hayashi, J. Takahashi, H. Fuji, and T. Aoki. *Constructive Side-Channel Analysis and Secure Design: 5th International Workshop, COSADE 2014,*

Paris, France, April 13-15, 2014. Revised Selected Papers, chapter A multiple-fault injection attack by adaptive timing control under black-box conditions and a countermeasure, pages 214–228. Springer International Publishing, Cham, 2014.

[20] S. Endo, N. Homma, Y.-i. Hayashi, J. Takahashi, H. Fuji, and T. Aoki. A multiple-fault injection attack by adaptive timing control under black-box conditions and a countermeasure. In *Constructive Side-Channel Analysis and Secure Design*, pages 214–228. Springer, 2014.

[21] N. F. Ghalaty, B. Yuce, M. Taha, and P. Schaumont. Differential Fault Intensity Analysis. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 49–58. IEEE, sep 2014.

[22] N. F. Ghalaty, B. Yuce, M. Taha, and P. Schaumont. Differential fault intensity analysis. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on*, pages 49–58, Sept 2014.

[23] B. Gierlichs, L. Batina, B. Preneel, and I. Verbauwhede. Revisiting higher-order dpa attacks: Multivariate mutual information analysis. Cryptology ePrint Archive, Report 2009/228, 2009. `http://eprint.iacr.org/2009/228`.

[24] B. Gierlichs, J.-M. Schmidt, and M. Tunstall. Infective Computation and Dummy Rounds: Fault Protection for Block Ciphers without Check-before-Output. pages 305–321. Springer Berlin Heidelberg, 2012.

[25] J. D. Golić and C. Tymen. Multiplicative Masking and Power Analysis of AES. pages 198–212. Springer, Berlin, Heidelberg, 2003.

[26] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi. A testing methodology for side-channel resistance validation. NIST Non-Invasive Attack Testing Workshop (NIAT 2011), September 2011. `http://csrc.nist.gov/news_events/non-invasive-attack-testing-workshop/papers/08_Goodwill.pdf`.

[27] A. Gornik, A. Moradi, J. Oehm, and C. Paar. A Hardware-Based Countermeasure to Reduce Side-Channel Leakage: Design, Implementation, and Evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1308–1319, aug 2015.

[28] S. Guilley, S. Chaudhuri, L. Sauvage, T. Graba, J.-L. Danger, P. Hoogvorst, V.-N. Vong, M. Nassar, and F. Flament. Shall we trust WDDL? In *Future of Trust in Computing*, pages 208–215. Vieweg+Teubner, Wiesbaden, 2009.

[29] Y. Ishai, A. Sahai, and D. Wagner. Private Circuits: Securing Hardware against Probing Attacks. pages 463–481. Springer, Berlin, Heidelberg, 2003.

[30] M. Joye, P. Paillier, and B. Schoenmakers. On Second-Order Differential Power Analysis. pages 293–308. Springer, Berlin, Heidelberg, 2005.

[31] D. Karaklajic, J. Schmidt, and I. Verbauwhede. Hardware Designer's Guide to Fault Attacks. *IEEE Trans. VLSI Syst.*, 21(12):2295–2306, 2013.

[32] R. Karri, G. Kuznetsov, and M. Goessel. Parity-based concurrent error detection of substitution-permutation network block ciphers. In *Cryptographic Hardware and Embedded Systems-CHES 2003*, pages 113–124. Springer, 2003.

[33] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. pages 388–397. Springer, Berlin, Heidelberg, 1999.

[34] Y. Li, K. Sakiyama, S. Gomisawa, T. Fukunaga, J. Takahashi, and K. Ohta. Fault Sensitivity Analysis. pages 320–334. Springer, Berlin, Heidelberg, 2010.

[35] M. Medwed and J. M. Schmidt. A Generic Fault Countermeasure Providing Data and Program Flow Integrity. In *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC '08. 5th Workshop on*, pages 68–73, Aug 2008.

[36] T. S. Messerges. Using Second-Order Power Analysis to Attack DPA Resistant Software. pages 238–251. Springer, Berlin, Heidelberg, 2000.

[37] N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, and E. Encrenaz. Experimental evaluation of two software countermeasures against fault attacks. In *Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on*, pages 112–117, May 2014.

[38] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson. Formal verification of a software countermeasure against instruction skip attacks. Cryptology ePrint Archive, Report 2013/679, 2013. `http://eprint.iacr.org/`.

[39] C. O'Flynn and Z. D. Chen. *ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research*, pages 243–260. Springer International Publishing, Cham, 2014.

[40] S. Patranabis, A. Chakraborty, and D. Mukhopadhyay. Fault Tolerant Infective Countermeasure for AES. In *Security, Privacy, and Applied Cryptography Engineering*, pages 190–209. Springer, 2015.

[41] C. Patrick, Y. Yao, and P. Schaumont. Addroundkey power model for fault injection enabled cpa attacks. Cryptology ePrint Archive, Report 2017/xxx, 2017. `http://eprint.iacr.org/2017/xxx`.

[42] C. Patrick, B. Yuce, N. F. Ghalaty, and P. Schaumont. Lightweight fault attack resistance in software using intra-instruction redundancy. Cryptology ePrint Archive, Report 2016/850, 2016. `http://eprint.iacr.org/2016/850`.

[43] T. Roche, V. Lomné, and K. Khalfallah. Combined Fault and Side-Channel Attack on Protected Implementations of AES. pages 65–83. Springer, Berlin, Heidelberg, 2011.

[44] J. M. Schmidt and C. Herbst. A Practical Fault Attack on Square and Multiply. In *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC '08. 5th Workshop on*, pages 53–58, Aug 2008.

[45] P. Schwabe and K. Stoffelen. All the aes you need on cortex-m3 and m4. Cryptology ePrint Archive, Report 2016/714, 2016. `http://eprint.iacr.org/2016/714`.

[46] A. Singh, M. Kar, A. Rajan, V. De, and S. Mukhopadhyay. Integrated all-digital low-dropout regulator as a countermeasure to power attack in encryption engines. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 145–148. IEEE, may 2016.

[47] M. Taha, A. Reyhani-Masoleh, and P. Schaumont. Stateless Leakage Resiliency from NLFSRs. In *IEEE Symposium on Hardware Oriented Security and Trust (HOST 2017)*, 2017.

[48] K. Tiri, D. Hwang, A. Hodjat, B.-C. Lai, S. Yang, P. Schaumont, and I. Verbauwhede. Prototype IC with WDDL and Differential Routing DPA Resistance Assessment. pages 354–365. Springer Berlin Heidelberg, 2005.

[49] E. Trichina. Combinational logic design for aes subbyte transformation on masked data. Cryptology ePrint Archive, Report 2003/236, 2003. `http://eprint.iacr.org/2003/236`.

[50] H. Tschofenig and M. Pegourie-Gonnard. NIST Lightweight Cryptography Workshop 2015, 2015. `https://www.nist.gov/news-events/events/2015/07/lightweight-cryptography-workshop-2015`.

[51] M. Tunstall and D. Mukhopadhyay. Differential fault analysis of the advanced encryption standard using a single fault. Cryptology ePrint Archive, Report 2009/575, 2009. `http://eprint.iacr.org/2009/575`.

[52] M. Tunstall and D. Mukhopadhyay. Differential fault analysis of the advanced encryption standard using a single fault. Cryptology ePrint Archive, Report 2009/575, 2009. `http://eprint.iacr.org/`.

[53] J. Waddle and D. Wagner. Towards efficient second-order power analysis. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, pages 1–15, 2004.

[54] C. Wolf. Picorv32 - a size-optimized risc-v cpu. `https://github.com/cliffordwolf/picorv32`, 2017.

[55] W. Yu, O. A. Uzun, and S. Köse. Leveraging on-chip voltage regulators as a countermeasure against side-channel attacks. In *Proceedings of the 52nd Annual Design Automation Conference on - DAC '15*, pages 1–6, New York, New York, USA, 2015. ACM Press.

[56] B. Yuce, N. F. Ghalaty, C. Deshpande, C. Patrick, L. Nazhandali, and P. 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 on - HASP 2016*, pages 1–8, New York, New York, USA, 2016. ACM Press.

[57] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont. Software Fault Resistance is Futile: Effective Single-Glitch Attacks. In *2016 Workshop*

on Fault Diagnosis and Tolerance in Cryptography (FDTC), pages 47–58. IEEE, aug 2016.

[58] Y. Zhang, G. Wang, Y. Ma, and J. Li. A Comprehensive Design Method Based on WDDL and Dynamic Cryptosystem to Resist DPA Attack. In *2011 International Conference on Intelligence Science and Information Engineering*, pages 333–336. IEEE, aug 2011.