Towards Lightweight Cryptographic Primitives with Built-in Fault-Detection

Thierry Simon[1], Lejla Batina[1], Joan Daemen[1], Vincent Grosso[2], Pedro Maat Costa Massolino[1], Kostas Papagiannopoulos[1], Francesco Regazzoni[3], and Niels Samwel[1]

[1] Digital Security Group, Radboud University, The Netherlands
thierry.simon.13@gmail.com,
{lejla,joan,P.Massolino,k.papagiannopoulos,n.samwel}@cs.ru.nl
[2] CNRS/Hubert Curien Laboratory, St.-Etienne, France
vincent.grosso@rub.de
[3] ALaRI, University of Lugano, Switzerland
regazzoni@alari.ch

Abstract. We introduce a novel approach for designing symmetric ciphers to resist fault injection. The approach is fairly generic and applies to round functions of block ciphers, cryptographic permutations and stream ciphers. We showcase our method with a new permutation called Frit and perform fault analysis on a simulated hardware and actual software implementation. We present performance results for software and hardware implementations with and without the fault detection mechanism. On a Cortex-M4 platform the overhead of the countermeasure in cycles is 83%. The penalty on resources for hardware implementations depends on the hardware and can be as low as 56%.

Keywords: design of cryptographic primitives, fault injection countermeasures, side-channel attack, lightweight implementations

1 Introduction

Today's world is digital and our daily business relies on the devices that we carry on us such as bank and transportation cards, car keys, phones and other embedded and mobile device. Many of these devices have to operate with very low power or energy and the continuous shrinking of the cores puts challenging constraints on area and memory.

Additionally, they are exposed to attackers that may exploit physical leakages of the cryptographic computation to extract cryptographic keys and other private data. This physical leakage can take the form of computation time, power consumption, electromagnetic radiation or other physical emanations and their exploitation is called side-channel analysis (SCA) [20,19]. The area of SCA has become an active field of research with advances in attacks and countermeasures. Another vulnerability of these devices is fault analysis, where an attacker provokes faults in the cryptographic computation and uses the resulting outputs to

recover the key. Boneh et al. showed in [8] that only a single computation fault is sufficient to extract an RSA private key. Like SCA, fault analysis (FA) has become a rich research field and designers and implementers have started working together to come up with secure designs and implementations of cryptographic primitives.

In the academic world, this need for low-resource cryptography with exposure to SCA and FA has given rise to the subfield of lightweight cryptography. This discipline has primarily focused on the design of small-size block ciphers with a round function that can be implemented compactly. Examples include PRESENT [7], KATAN and KTANTAN[13] and more recently GIFT, an update of PRESENT based on more modern insights [2]. Recent proposals have addressed tweakable block ciphers and here Skinny [5] is a prominent example. Other avenues included low-latency block ciphers like PRINCE [9] and low-latency tweakable block ciphers like Mantis [5] or low-energy block ciphers like MIDORI [1]. The (tweakable) block cipher approach has the advantage that the modes on top of the primitive for encryption, authentication and key derivation become simpler than with simple block cipher. However, dedicated functions may lead to more efficient solutions. For example, the MAC function Chaskey [23] is a dedicated MAC function with a very small workload on certain low-end CPUs. More recently, the insight that one can do efficient encryption and authentication with the sponge and duplex construction calling a cryptographic permutation [**?**] had led to the emergence of several lightweight permutations such as Gimli [6] and Mixifer [30]. The relative large width of these permutations as compared to lightweight block ciphers is compensated by the low overhead of the the duplex and sponge modes, leading to a total solution taking significantly less resources than a block cipher based solution.

The majority of lightweight designs do not take into account resistance against SCA and/or FA. These are often considered issues that should be dealt with in the actual implementation. However, some designs did take it into account by using non-linear components that lend themselves to masking. In this category we can mention Noekeon [12], the LS-Designs [16], FANTOMAS, ZORRO and ROBIN and also the Keccak-f permutation [25].

Hence, designers have started already more than a decade ago looking into special designs dedicated to the optimization of specific parameters (e.g. latency, power, energy) or introducing additional features like SCA/FA countermeasures etc. However, as far as we know, no proposals have taken into account the ability to provide protection against fault attacks from the design phase.

Recently, there were proposals for combined countermeasures against SCA and FA on existing ciphers, namely ParTI [28] and CAPA [27]. Although the contributions are relevant when aiming at crypto devices secured against physical attacks, the results incur a substantial overhead in area and performance. Providing protection against both SCA and faults is not easy in the sense that FA countermeasures take redundant representations and computations that increase total power consumption and EM leakage.

1.1 Related Work

Several works considered improving SCA resistance from the design phase, most notably Grosso et al. [16]. The authors study possible optimizations when specializing the designs to Boolean masking.

To our best knowledge there are only two works considering combined countermeasures against SCA and FA. Schneider et al. [28] introduce a countermeasure for cryptographic hardware implementations that combines the concept of threshold implementation with an error detecting approach against fault injection. The idea is demonstrated on the lightweight cipher LED and the total area of the ParTI protected cipher is roughly 2.5 times larger than the unprotected version. On the other hand, the work of Reparaz et al. [27] proposes a countermeasure that claims security against higher-order SCA, multiple-shot DFA and combined attacks but it comes at price. A chosen set of parameters implies a protected hardware implementation of AES of 215 105 GEs. Both works have in common that they apply their strategies to existing ciphers.

With respect to SCA resistance a cautionary note is necessary. Regazzoni *et al.* [26] showed that, in the context of an AES S-box, various error detection mechanisms increase the vulnerability to power analysis attacks. In a similar fashion, Cojocar *et al.* [11] highlighted the trade-off between instruction duplication and side-channel resistance, using this time horizontal exploitation techniques. Our work shows that this is not always the case, i.e., there exist fault-resistant mitigations that do not diminish resistance to SCA. This fact has also been verified in this paper.

1.2 Our Contributions

In this work we present a design approach for building round functions that have built-in protection against fault injection attacks. Our approach is based on a redundant representation of the state and a round function extension with an invariant property. We identify a number of lightweight operations for nonlinearity and diffusion that are compatible with this approach. In contrast to the approaches in [28] and [27], we start from a novel approach leading to a better trade-off between resource usage and level of protection against cryptanalysis, SCA and fault attacks.

To demonstrate the approach we include an example cryptographic permutation called Frit (for Fault-Resistant Iterative Transformation) that has a competitive performance in both hardware and software. The approach is however flexible and can also be used to build (tweakable) block ciphers, stream ciphers or MAC functions.

We demonstrate the FA resistance of Frit when implemented on a 32-bit microcontroller and an ASIC platform, i.e. we show how Frit provides single-fault resistance at a reasonable overhead. Providing security against a more potent adversary with tampering capabilities [17] or with ineffective fault capabilities [14] usually results in much higher overhead and remains outside our current scope. We also show that the SCA resistance does not decrease as much, with the new

approach on FA mitigations, as with duplication. This is the first step towards adding a concrete SCA protection such as a TI scheme or some other masking approach.

1.3 Organization of this Paper

We present our general idea in Section 2. We illustrate the validity of our approach with a concrete lightweight cryptographic permutation that we present in Section 3. We motivate its design in Section 4, integrate the fault countermeasure in Section 5 and discuss its performance in Section 6. We report on fault experiments on both a simulated hardware and an actual software implementation in Section 7. Finally, we analyze the impact of redundant computations on DPA resistance in Section 8.

2 Building Fault-Resistant Round Functions

In this section we discuss operations that are useful in round functions and that lend themselves to fault detection at a relatively low cost.

The round functions we consider operate on a *state* that is partitioned in n equally sized *limbs*. For the purpose of fault detection, we extend this state with an additional limb and compute an *extended round function* on this extended state. This extended round function has two properties:

Correctness The effect of the extended round function on the first n limbs is the same as the round function itself.

Sum-invariance Let the *limb-sum* of the extended state be the bitwise sum (XOR) of the $n+1$ limbs. The extended round function preserves the limb-sum.

So the $n+1$-th limb can be seen as a simple checksum of the state as it is the bitwise sum of its n limbs plus a constant. The value of this constant is the initial value of the limb-sum.

In order to build a sum-invariant round function, we simply compose it from sum-invariant steps. For the step functions, we can typically distinguish four operations, each with its own function: non-linear operations, mixing operations, transpositions and round key addition. For example in AES these map to the four step functions SubBytes, MixColumns, ShiftRows and AddRoundKey. Clearly, in the extended round function achieving sum-invariance may induce overhead processing in each of the steps . The key idea of our approach is to choose the specific operations to make that overhead small, hereby making use of the following facts:

limb transposition This is a re-ordering of the native limbs. It does not impact the limb-sum and hence has no overhead.

limb adaptation When modifying one limb by bitwise adding a function f of the limbs, limb-sum variance just requires computing that function twice and also adding it to the checksum limb. adding sum-invariance doubles the computational cost.

limb-sum switch Replacing the first limb by the bitwise sum of all limbs can be achieved with just a transposition on the extended state and adding a constant to two limbs. Namely, the checksum limb already contains this bitwise sum plus a constant, so it suffices to switch the first limb and the checksum limb and to add the initial value of the limb-sum to both. So here the computation on the extended state is actually lighter than the computation on the native state.

The simplest invertible non-linear function is the addition to a limb of the bitwise AND of two other limbs. For the mixing layer, one can add to a limb two cyclically rotated versions of the same limb. This achieves mixing between bits that are in different positions in the limbs. Replacing the first limb by the bitwise sum of all other limbs achieves mixing between bits of different limbs.

The protection offered by the state and round function extension is that a fault in the computation is likely to change the limb-sum. Such a change can be detected after any round by computing the limb-sum and comparing that with the (stored) initial value of the limb-sum. Non-equality means there was a fault, but equality does not necessarily mean there has been no fault. In the latter case we speak of *undetected faults*. Clearly, a fault in a single limb, whatever its Hamming weight, will affect the limb-sum. Note, furthermore, that in this case the sum-invariance property will ensure that the limb-sum remains invalid through the following steps. The only way the limb-sum can return to the valid value is by another fault, that must exactly compensate for the first fault.

Our primary goal is the guaranteed detection of any single fault in the computation of the datapath. For this, the computation of f in limb adaptation steps must be done twice, rather than adding the result of the same computation to both the native limb and the checksum limb. The easiest attack with this double computation in place would be to inject the same fault in each of the two computations, so that they compensate each other in the limb-sum. In this respect it is a good idea to use different registers and/or different computational sequences (in software) or different combinatorial circuits for the two computations, so that the attacker has to induce two different faults that have the same net effect. Alternatively, an attacker could also inject, in between operations, compensating faults on two limbs that would leave the limb-sum of the state unchanged. To be successful, such an attack would require knowledge of the implementation details and the ability to very precisely inject faults.

However, there are other types of faults that are not covered by the countermeasures and they must be countered with other means. For example, skipping a full step, round or number of rounds, will not affect the limb-sum. Another example are faults in the limb-sum checking mechanism itself, e.g., just faulting the reporting of the comparison outcome from *different* to *equal*. Clearly, implementations must have some redundancy in the control flow logic for the handling of the steps and the limb-sum checking.

3 FRIT Design and Specification

In order to illustrate our design approach, we present Frit, a 384-bit cryptographic permutation, whose building blocks consist in the operations introduced in Section 2. Not only do they make Frit lightweight, but they also allow for a cheap extension of the permutation.

Algorithm 1 Frit

Input: $a, b, c \in \{0,1\}^{128}$
Output: $(e, f, g) = \text{FRIT}(a, b, c)$
for i from 0 to 15 inclusive **do**
  $c \leftarrow c \oplus \text{RC}_i$
  $a \leftarrow a \oplus (a \lll 110) \oplus (a \lll 87)$
  $c \leftarrow c \oplus (a \wedge b)$
  $c \leftarrow c \oplus (c \lll 118) \oplus (c \lll 88)$
  $b \leftarrow a \oplus b \oplus c$
  $(a, b, c) \leftarrow (c, a, b)$
end for
return (a, b, c)

Notation. Frit operates on a state of three limbs in $\{0,1\}^{128}$, the set of bitstrings of length 128. We will denote by

- RC_i, the i-th round constant,
- $x \oplus y$, the exclusive *or* (XOR) of limbs x and y,
- $x \wedge y$, the bitwise logical *and* of limbs x and y,
- $x \lll k$, the bitwise rotation to the left of limb x by factor k.

The permutation is composed of 16 identical rounds applied on a state formed by three limbs a, b and c. Each round, represented in Fig. 1, has 6 steps: (1) the round constant addition; (2) a first mixing step; (3) a Toffoli gate [31]; (4) a second mixing step; (5) a switch operation; (6) a transposition.

Round Constants Similarly to what is done for *Mixifer* [30], a master round constant is generated from a linear-feedback shift register. Choosing feedback polynomial $1 + x^2 + x^5$ and initial state $0b11111$, we get the bit sequence $0b11111001101001000010101110110001$ as master round constant. The round constant for round i is then obtained by shifting the master round constant by i bits to the right.
At the start of the i-th round, the round constant is added to limb c. In particular, this operation can be written as $R_i(a, b, c) = (a, b, c \oplus \text{RC}_i)$.

Mixing Steps Each round contains two mixing steps M_i with $i = 1, 2$ defined respectively by $M_1(a, b, c) = (a \oplus (a \lll 117) \oplus (a \lll 87), b, c)$ and $M_2(a, b, c) = (a, b, c \oplus (c \lll 118) \oplus (c \lll 88))$. Their main purpose is to

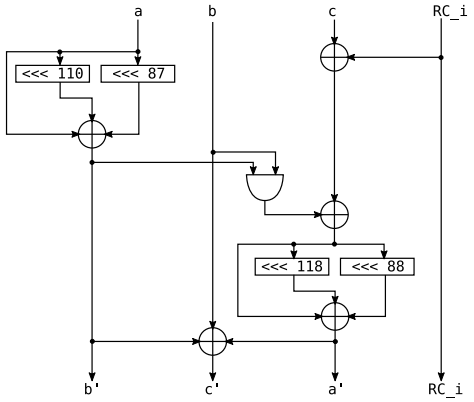


Fig. 1. Round of the permutation

achieve fast diffusion. The value of the rotation offsets are justified in Section 4. Note that the first mixing step can be computed in parallel to the round constant addition, as both operations act on different limbs.

Toffoli Gate The Toffoli gate is defined by $T(a, b, c) = (a, b, (a \wedge b) \oplus c)$. It is the only non-linear operation in the permutation.

Switch. The three limbs are summed together as elements of $\mathbb{F}_{2^{128}}$. Concretely, the operation is given by $S(a, b, c) = (a, a \oplus b \oplus c, c)$.

Transposition The operation is defined by $\tau(a, b, c) = (c, a, b)$.

4 Propagation Analysis

In this section we analyze the propagation properties of Frit and its resistance to invariant attacks. Furthermore, we used this analysis to choose the rotation offsets in the mixing steps and the round constants. We are well aware that the propagation analysis is not enough to motivate the security of Frit. With that in mind, we intend to provide some differential and linear cryptanalysis in a future version of this paper. Moreover, we would like to provide some algebraic analysis, that goes deeper than the few properties we mention in appendix A.

4.1 Avalanche Tests

A property that is very informative about the vulnerability of a cryptographic primitive against structural distinguishers such as impossible differentials, integral cryptanalysis or truncated differentials is *full diffusion*. We say a cryptographic permutation achieves full diffusion if for a random input, the change of any of the input bits affects all bits of the output. If a cryptographic permutation achieves full diffusion after n rounds, it is likely that it does not have exploitable structural distinguishers covering more than $2n$ rounds. We gave here a rather informal definition of full diffusion. However, inspired by what is done

for Gimli [6], we tested our permutation against three different interpretations of the criterion, each with a more formal definition.

Full Dependency. In this test, we replace in the round function every exclusive *or* and bitwise logical *and* operation by a bitwise logical *or*. Then, we evaluate the number of rounds needed to transform a state of weight 1 — that is, with all the bits set to 0, except one bit set to 1 — into a state full of 1. The conclusion of this test is that at most 5 rounds are required to reach full dependency.

Avalanche Criterion. We evaluate the number of rounds required to observe the property that a single bit of the input flipped, results in half of the output bits flipped. To do that, we use a Monte-Carlo approach. We generate 1000 random input states of 384 bits and flip for every input the bit at each position once. We then observe that 7 rounds are needed to meet the avalanche criterion. Detailed results of the average number of flipped bits resulting from a flipped bit at any position in the input after 7 rounds are given in Appendix D. Table 1 gives the average Hamming distance between output and input states over the 384 000 experiments, as well as the associated standard deviation, in terms of the number of rounds. While those results do not take the position of the flipped bit into account, they are a good indication that 7 rounds are needed to flip half the bits.

Round	Expected Hamming distance	Deviation
1	53	49.56
2	101.87	79.87
3	124.65	65.12
4	159.67	36.51
5	185.96	15.88
6	191.59	10
7	192.08	9.72
8	192.06	9.84
9	191.9	9.8

Table 1. Expected Hamming distance between output and input states

Strict Avalanche Criterion. The avalanche criterion describes how a flipped bit affects the entire output of the permutation. However, it does not say anything about how each individual bit of the output is affected by such a change. Therefore we also considered a stronger test, that consists in evaluating the probability p_{ij} of observing a flipped bit at position j if bit i is flipped. The strict avalanche criterion is then satisfied if those probabilities are close to 0.5 for all possible positions. Concretely, we generated 100 000 random input states of 384 bits and flipped the bit at each position once. Again, we observed that only 7 rounds are needed to have all the p_{ij} comprised between 0.49 and 0.51.

Those tests helped us establish the structure of the permutation as well as fix the rotation offsets and choose the number of rounds. In fact, the rotation offsets of Frit were chosen to achieve diffusion and the avalanche effects in the smallest number of rounds. Note that other tuples of offsets satisfy the tests in the same amount of rounds. Since full diffusion is achieved in 7 rounds, we fixed the number of rounds for Frit as two times this amount plus a small margin, i.e. 16 rounds.

4.2 Invariants Analysis

As rotation offsets influence the propagation properties of the permutation, so do the round constants influence its invariance properties. Indeed, round constants, if properly chosen, break the symmetries of the permutation and can defeat slide attacks, invariant subspace attacks and non-linear invariant attacks [4]. Let D be the set of differences of the round constants $RC_i \oplus RC_j$ and let L be the linear layer of Frit, i.e. the round of the permutation without the round constant addition and the Toffoli gate. Then, we can verify that the smallest L-invariant subspace of $\mathbb{F}_{2^{384}}$ that contains D is of maximal dimension. In particular, [4, Prop. 2] implies that Frit is resistant to invariant attacks.

5 Integration of Fault Detection

This section describes concretely how the fault countermeasures introduced in Section 2 are integrated to Frit. In order to extend the state of the permutation, a fourth 128-bit limb d is added. This limb is initialized to $a \oplus b \oplus c$, so that the limb-sum of the initial state equals zero. The round function is then extended on $\{0,1\}^{512}$ by extending each of its steps. Let P be one of the previously defined steps, its extension $\bar{P}(a,b,c,d) = (a',b',c',d')$ is chosen such that

$$(a',b',c') = P(a,b,c), \tag{1}$$

$$a' \oplus b' \oplus c' \oplus d' = a \oplus b \oplus c \oplus d, \tag{2}$$

where conditions 1 and 2 stand respectively for correctness and sum-invariance.

The round constant addition, mixing steps and Toffoli gate operation are extended using the limb adaptation technique. As illustrated in Fig. 2 and 3 for the first mixing step and the Toffoli gate, it consists in computing the operation a second time on the redundant limb for a computational overhead of a factor two. The extended operations are given respectively by:

- $\bar{R}_i(a,b,c,d) = (a,b,c \oplus \mathrm{RC}_i, d \oplus \mathrm{RC}_i)$,
- $\bar{M}_1(a,b,c,d) = (a \oplus (a \lll 110) \oplus (a \lll 87), b, c, d \oplus (a \lll 110) \oplus (a \lll 87))$,
- $\bar{M}_2(a,b,c,d) = (a,b,c \oplus (c \lll 118) \oplus (c \lll 88), d \oplus (c \lll 118) \oplus (c \lll 88))$,
- $\bar{T}(a,b,c,d) = (a,b,(a \wedge b) \oplus c, (a \wedge b) \oplus d)$.

The switch and transposition operations are extended using respectively the limb-sum switch and limb transposition techniques for no computational overhead. The extended operations are given by:

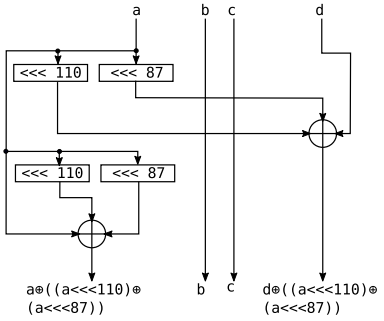


Fig. 2. Extended mixing step

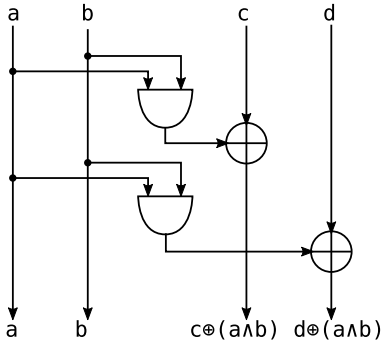


Fig. 3. Extended Toffoli gate

- $\bar{S}(a, b, c, d) = (a, b, d, c)$,
- $\bar{\tau}(a, b, c, d) = (c, a, b, d)$.

Fig. 4 represents a round of Frit integrating the redundant representation.

Checking for faults can be done after any step of the round function and consists in verifying whether the limb-sum of the state equals its initial value. While checking after each step or after each round would make it possible to detect faults occurring respectively during different steps or rounds, it would also result in a big performance penalty. Furthermore, such frequent checks would still be unable to detect compensating faults within the same step of the round function. Therefore, we recommend a single check at the end of the permutation. That means that, in the entire permutation, only a single fault — i.e. a fault in only a single word — is certain to be caught. Multiple random faults are likely to be caught, but the countermeasure would be ineffective against an adversary able to inject multiple faults with high precision.

5.1 Cost Analysis and Scalability

The countermeasure introduced in Section 2 detects single faults in the computation of Frit. We emphasize that it uses the parity code [4,3,1] over $\mathbb{F}_{2^{128}}$. Other linear codes could detect more fault injections at the cost of increasing the code's minimal distance. For unprotected permutations with more limbs than Frit, codes of higher dimension would be required.

The computational overhead of a $[n, k, d]$ code used on an unprotected implementation is given by a factor $(n-k)+1$, in the case of a systematic code. We can see that this estimation is tight for the limb adaptation, but too pessimistic for the limb transposition and the limb-sum switch. The limb-sum switch is particularly adapted to our [4,3,1] code, but if we have different operations other codes may be more efficient.

In Table 2 we give figures for the full Frit overhead of the countermeasure in terms of the number of bitwise operations on 128-bit limbs and compare

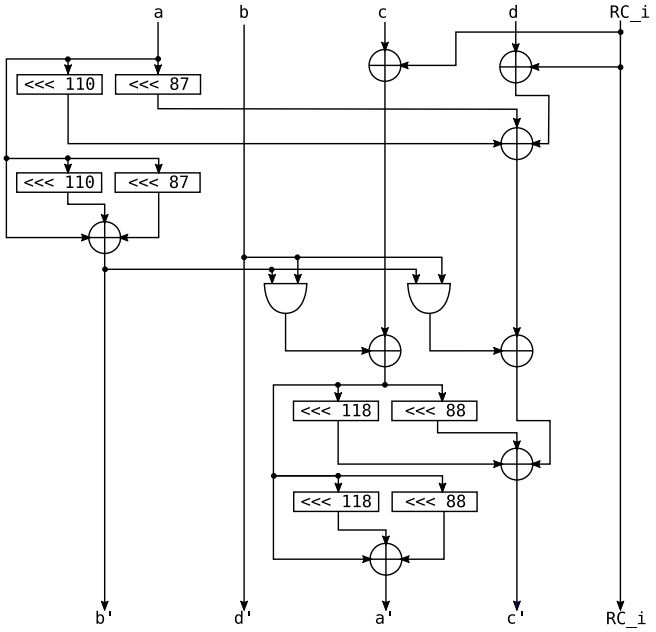


Fig. 4. Extended round of FRIT

with duplication that should have similar specification. The duplication code is a $[2n, n, 1]$ code. While it has the same detection capability as our $[n + 1, n, 1]$ parity code, it also requires a state that is almost twice as big and its overhead of a factor 2 is tight for all operations. Additionally, we observe that the number of 32-bit registers required to store the entire state only increases by 25% with our countermeasure against 100% for duplication.

Countermeasure	**Bitwise Operations**			**Number of**
	XOR	***Rotation***	***AND***	**32-Bit Registers**
None	128[†]	64	16	12
This work	194[‡]	128	32	16
Duplication	256	128	32	24

Table 2. Cost of the countermeasures

[†] The addition of the 32-bit round constant is here considered as a full 128-bit XOR.
[‡] Here we add to the twelve XOR per round the two needed to initialize limb d.

6 Implementation Results

6.1 Hardware

Frit is specified in terms of bitwise AND and XOR operations and can thus be implemented straightforwardly in hardware. We construct a simple round-based architecture and variant with combinatorial logic that implements a number of rounds dividing the total number with 16. Each round is first implemented with the countermeasure shown in Figure 4 and then without the countermeasure as in Figure 1. For each round architecture, with and without countermeasure, we differentiate among 5 cases: 1, 2, 4, 8 and 16 rounds. We call the latter the *full-round* version. All these 10 versions are wrapped into a similar high-level architecture depicted in Figure 5 that takes care of the input, the output, initialization of register d and the final check of the limb-sum for fault detection. The architecture stores the state in an input/output shift register. This allows serialization of external communication through an 8 bit bus, while internally load and store operations can be done on the full state in parallel. The initialization of register d is performed with a bitwise XOR operation on the initial state values of registers a, b and c and the final limb-sum check at the end of the sixteenth round that compares the XOR of the four limbs with 0.

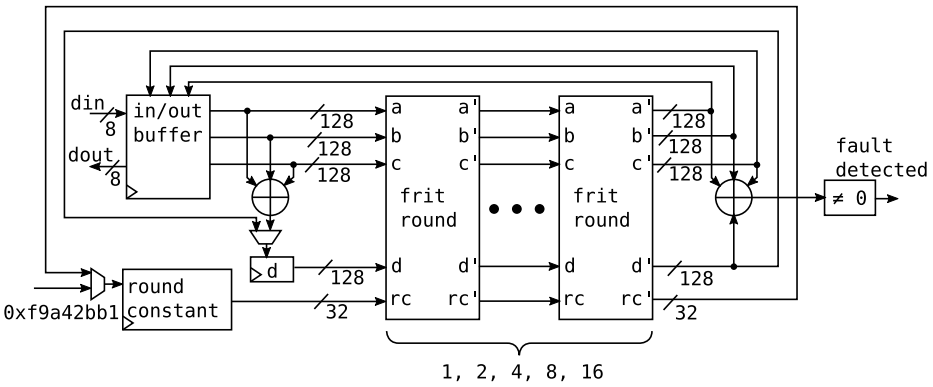


Fig. 5. High level architecture. For the 16 rounds version, the registers d and *round constant* are removed and their inputs are fed directly. Also, in the version without countermeasure, the register d and the check are also not present.

The full-round version was optimized according to the architecture in Figure 5. In particular, there is no need for feeding back the output of the round logic to its input and the round constants can be hard-coded in the logic. In the version with countermeasure, no dedicated register is needed for limb d.

Table 3 shows the implementation results for the different options after place and route, on Xilinx Vivado 2017.4.1. The FPGA chosen is the Kintex-7

xc7k160tfbg676, a model that belongs to the SAKURA-X board. [29]. All results were obtained with the default tool settings.

The results in Table 3 show that Frit is a competitive permutation with and without the countermeasure when compared with Gimli, 12 rounds of Keccak-f[400], and the software-oriented permutations Salsa20/20 and Chacha20/20. Comparing the results with and without the countermeasure yields a difference of 56% in case for the 1 round per cycle version, aka Frit(C)–1, when taking the slices into account. However, this difference increases to 164% for the 16 rounds per cycle version, aka Frit(C)–16. Minimizing the optimizer — represented in the table by the entry Frit(C)–16 (No opt.) — reduces this difference to 141%. From this, we conclude that the increase in the overhead is not exclusively related to the optimizer, but also to the routing of the tool and the countermeasure cost.

Table 3 shows some unexpected result, as Frit(N)–4 appears to be cheaper than Frit(N)–2. One possible explanation for this phenomenon is that the tool optimizer was able to take advantage of some structure present in this version to make a very small circuit.

Permutation	State size	Cycles	Resources	Period (ns)	Time (ns)	Res.×Time/ State
Gimli	384	1	2248 S(8687 L+401 F)	21.909	22	128.3
Keccak-f[400] 12r	400	1	2993 S(11491 L+417 F)	31.971	32	239.2
Salsa20/20	512	1	4128 S(16141 L+521 F)	210.266	211	1695.3
Chacha20/20	512	1	3346 S(13137 L+521 F)	152.720	153	998.0
Frit(C)–16	384	1	2500 S(9289 L+402 F)	29.923	30	194.8
Frit(N)–16	384	1	947 S(3514 L+401 F)	30.579	31	75.4
Frit(C)–8	384	2	1267 S(4838 L+544 F)	16.844	34	111.2
Frit(N)–8	384	2	566 S(2110 L+414 F)	15.198	31	44.8
Frit(C)–4	384	4	705 S(2672 L+558 F)	9.182	37	67.4
Frit(N)–4	384	4	372 S(1348 L+429 F)	7.247	29	28.1
Frit(C)–2	384	8	608 S(2326 L+555 F)	4.837	39	61.3
Frit(N)–2	384	8	441 S(1626 L+425 F)	4.056	33	37.3
Frit(C)–1	384	16	341 S(1239 L+556 F)	3.955	64	56.2
Frit(N)–1	384	16	218 S(773 L+426 F)	2.146	35	39.4
Frit(C)–16 (No opt.)	384	1	2239 S(8472 L+392 F)	31.487	31	183.6
Frit(N)–16 (No opt.)	384	1	930 S(3450 L+391 F)	31.296	31	75.8

Table 3. Results on FPGA Kintex 7 xc7k160tfbg676-1 for Frit permutation with countermeasure (C) and without (N) and others. Slice(S). LUT(L). Flip-Flop(F)

6.2 Software

We now describe an implementation of Frit optimized for speed on an embedded ARM Cortex-M4 microcontroller, with and without the fault detecting countermeasure.

We opted for a bitsliced representation of the state of the permutation. More precisely, every 128-bit limb x is represented as four 32-bit words x_0, x_1, x_2 and x_3, such that the word x_i contains the bits of x with indices equal to i modulo 4. This representation offers two main advantages. First, in the implementation without countermeasure, it makes it possible to fit the entire round computation in the 14 registers of the Cortex M4 that are free to use, i.e. registers $r0$ to $r12$ and $r14$. Indeed, 12 registers are required to store the 384-bit state of Frit with 2 additional registers left for temporary values. This means that intermediate results do not need to be stored and loaded from memory. This is however not the case when the state is extended with an additional limb. Second, the bitsliced representation allows for an efficient implementation of each mixing step at the cost of 8 XOR and 8 circular shift of registers. The use of the *barrel shifter*, a feature of the Cortex M4, reduces this cost further to only 8 XOR. The only downside of the bisliced representation is that it requires four additions of 8-bit round constants per round instead of the addition of a single 32-bit constant. However, this penalty is somewhat mitigated by the fact that, contrarily to 32-bit constants, the M4 is able to manipulate 8-bit constants without having to load them from memory.

Additionally, to save a few function calls, we unrolled the entire 16 rounds of the permutation. Our implementation of Frit takes then 1468 cycles without fault-resistance and 2682 with it, resulting in an overhead of about 83% for our countermeasure. This is considerably less than the 100% overhead we would have by using duplication. Thus, considering fault-resistance early on in the design phase of the cipher can improve performance, compared to a direct application of software countermeasures to e.g. AES [3].

7 Fault-Resistance Evaluation

In this section we present the practical evaluation of fault-resistance of Frit. We firstly discuss the robustness of a simulated hardware implementation that we evaluated with a logical simulator. We then discuss the robustness of the software implementation that we evaluated on a real device.

In our experiments, we identified four different reactions of the implementation to the fault injection:

- Normal: the fault attempt does not affect the correctness of the results. This is mainly the case when the glitch is too short and thus incapable of producing the effects wanted by the attacker.
- Reset: the fault is so disruptive that the device or simulation stops functioning. This is for instance the case when the fault hits the control unit.
- Success: the fault results in an incorrect result and is not detected. This type of fault happens, for instance, when the adversary hits exactly the same bits of two limbs.
- Detected: the fault is injected and detected. Most of the random faults (being single or multiple bits) belong to this category.

7.1 Attacking the Simulated Hardware Implementation

Figure 6 shows the flow we used to evaluate the fault-resistance of hardware implementation. The first step of the flow is the functional test of the design that we carried out using the logic simulator (in our case Modelsim 10.4d) and dedicated test benches. The functional verification is carried out using the test vectors reported in Appendix C. The evaluation of fault attack resistance can be carried out at different stages of the design flow, in the figure we report only the ones we used in this paper. The first stage is at the RTL level. At this stage, no information about timing delays or actual gates is available yet. However, an evaluation carried out at RTL level allows to confirm that the countermeasure behaves correctly, namely it is really capable of detecting an injected fault when it is supposed to. Furthermore, the analysis at RTL level is independent from the target platform where the design will be implemented (thus the behavioral analysis mimics both the behavior of ASIC or FPGAs). However, it does not give indications about the real feasibility of the attack or about the physical circuit that needs to be attacked. The second stage is after synthesis. In the figure we report the ASIC synthesis that we performed using the tools and the library discussed in Section 6. After synthesis, we know the exact gates composing the circuit and we have precise information about the delay of the gates. However, the results obtained at this stage are specific to the implementation and could be invalid when different technological libraries are used.

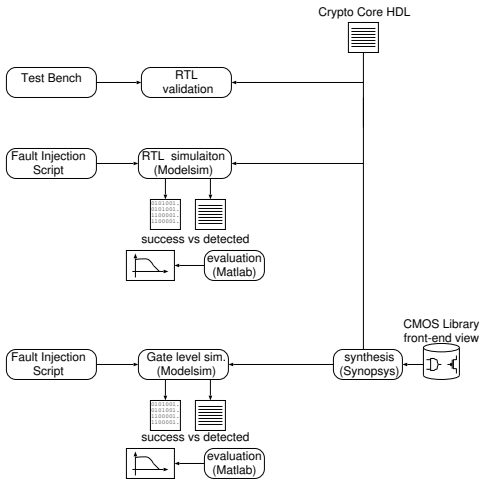


Fig. 6. Design and simulation flow for simulating fault injection in hardware

The fault injection is simulated by forcing a signal (or a set of signals) to a specific value, for a certain amount of time (up to the whole simulation time). With this approach, we can simulate glitches with a minimum granularity of one bit and a glitch minimal length equal to the time resolution of the simulation

tool. We evaluate the resistance of our designs injecting 100 000 faults, randomly inserted during the computation of the algorithm. The type of faults we injected included single-bit glitches (minimum length 8ns), single bit stuck-at, multiple bit glitch and stuck-at (up to 8 bits) on the same word, and multiple-bit glitch and stuck-at on different signals of the design (also in this case, we corrupted up to 8 bits). The same analysis was carried out at RTL level and on the netlist obtained after the synthesis. In both cases the designs were simulated with a clock period of 16ns.

The results of the simulation confirmed that the behavior of Frit is as expected, both at RTL level and after synthesis. An example of a *Success* fault and a *Detect* faults are reported in Figure 7 and Figure 8 respectively. The figures report the screen capture of the Modelsim simulation where the fault is injected. In the first case we can see the faulty signals (*temp_c(7)(119)* and *temp_d(7)(119)*) and the faulty detection signal (*test_fault_detected*) is low, showing that in this case, the fault is injected but not correctly detected. In the second case, we can see the faulty signal (*temp_c(7)(119)*) and the faulty detection signal is high, showing that the fault is detected correctly.

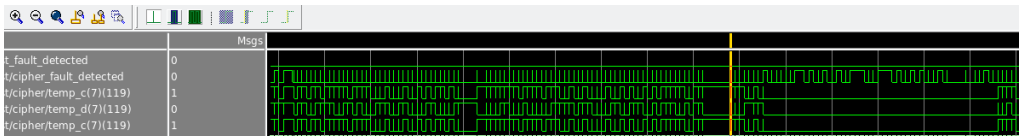


Fig. 7. Fault injection not detected.

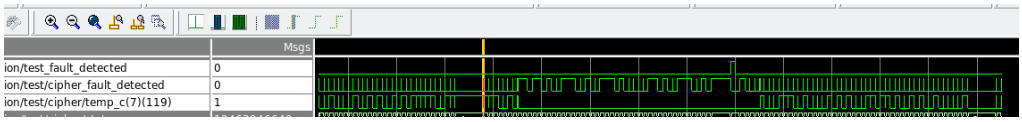


Fig. 8. Fault injection detected.

7.2 Attacking a Software Implementation

In this section we report on our fault injection experiments on a software implementation of the permutation. We apply a technique called electro-magnetic fault injection(EM-FI) to inject glitches. This is accomplished by emitting a short EM pulse from a specific location close to the target. Because this location is variable, it is possible to obtain very localized results.

Figure 9 shows a schematic overview of the setup. Our target is a development board containing a Cortex-M4F, more specifically the STM32F407IG. The PC is connected to the target with a serial connection to enable communication. The tools such as the VC Glitcher and the xy-table that we use are manufactured by Riscure[6].

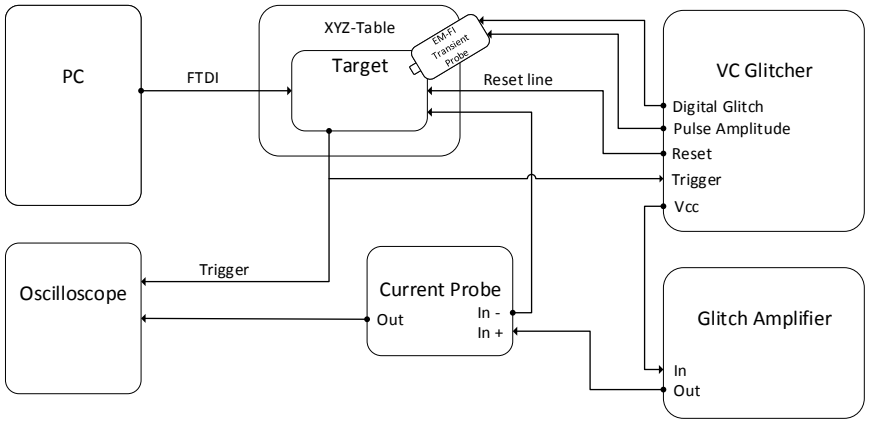


Fig. 9. This figure shows a schematic overview of all the components in the fault injection setup.

The xy-table moves a probe across the target with high precision. The VC Glitcher sends a signal so the probe will emit a pulse. The VC Glitcher also controls a reset line, in case the pulse was too strong and the board is unable to respond. An oscilloscope is used together with a current probe to measure the power consumption to determine a time window where the fault should be injected. Once this is found the oscilloscope and current probe are removed from the setup.

The PC is also connected to the oscilloscope to collect the measurements, as well as to the xy-table to position the probe and to the VC Glitcher to control the pulse strength and the timing. Figure 10 shows a photo of the setup with all the components described before.

Using the setup described earlier, we conducted an electro-magnetic fault injection experiment where we scanned the whole surface of a chip computing the last round of Frit. We divided the surface of the chip in a 100 by 100 grid and injected 10 glitches per position. This resulted in a total of 100 000 glitches. Table 4 shows the fault detection probability of the experiment. A glitch injected by this setup should result in at most a single fault. Out of the 100 000 glitches,

[6] `https://www.riscure.com/security-tools/hardware/`

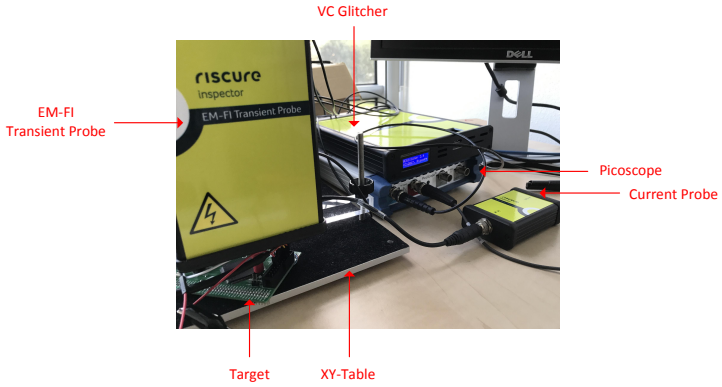


Fig. 10. A photo of the setup.

Table 4. Results for fault detection probability using 100 000 glitches.

Result	**Percentage**
Normal	62.46
Reset	30.45
Success	0.63
Detected	6.46

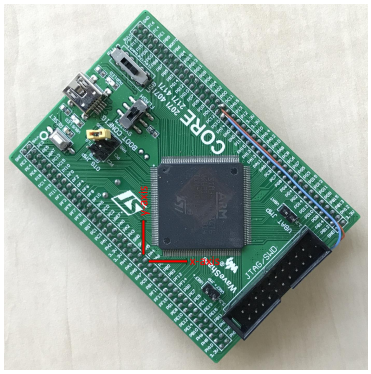
62.46% resulted in normal behavior, no fault occurred. The percentage is high because a large surface area of the chip remains unaffected by the glitches. In 30.45% of the glitches the glitch was too strong and the device stopped working and had to be reset to continue. A glitch that results in a fault requires glitch parameters that are on the border of normal and reset behavior, this is why many glitches resulted in a reset.

The table shows a small percentage of successful faults, i.e. faults producing an output that passes the check, but differs from the expected one. Eliminating duplicates within the successful faults reduced their number from 629 to 21. We then analyzed these undetected faults heuristically by inverting the permutation to study the intermediate states leading to the faulty outputs. Using this, we established that they were not applicable to the fault model of our countermeasure, since out of the 21 different successful faults:

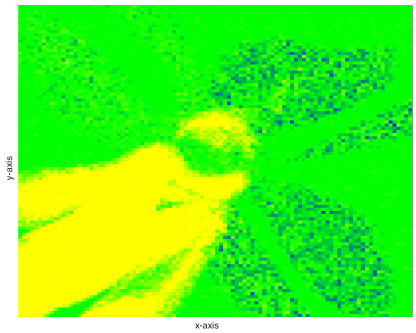
- 9 occurred during the checking phase,
- 5 modified the round constant when being loaded from memory,
- 4 swapped two limbs during the round,
- 1 zeroed out the entire state,
- 1 skipped the round function.

The remaining 6.46% of the glitches resulted in a fault and were detected. This means that over 90% of the glitches that resulted in a fault were detected by the scheme.

Figure 11a shows the target board. In Fig. 11b we see the results of the experiment in a heat map corresponding to the dimensions of the board, where each data point represents the average result of 10 glitches. The amount of successful faults that were found inapplicable is not sufficient to be visible in the figure.



(a) Target Board



(b) Heat map of the results of the following faults, Normal (green), Reset (yellow), Detected (blue).

Fig. 11.

8 SCA Evaluation

To offer a holistic security analysis of Frit, it is necessary to assess its security with respect to side-channel analysis in a concrete manner. From the viewpoint of a side-channel adversary, any form of repetitions or redundancy increases the exploitable information. Thus in the case of our fault-detecting permutation, the redundant 128-bit limb can enhance the available leakage and make key recovery easier. Regazzoni *et al.* [26] were among the first to analyze the interaction between fault injection countermeasures and side-channel attacks, focusing on parity-based error detection for AES. Using a similar approach Cojocar *et al.* [11] investigated the effect of instruction duplication and infective countermeasures on the side-channel leakage. Notably, both works stress that standard side-channel attacks such as univariate correlation power analysis are often incapable of exploiting all the available leakage when redundancy is present. Since such naive attacks are not able to reveal the full picture, they may lure the evaluator into a false sense of security, a fact that exacerbates the need for more concrete evaluation tools. In this direction, Veyrat-Charvillon *et al.* [32], as well as Le Bouder *et al.* [22], employed Soft Analytical Side-Channel Attacks

(SASCA), a new type of side-channel attack based on the Belief Propagation (BP) algorithm, in order to attack AES effectively. This particular attack has the advantage of horizontality, i.e. it is able to exploit the entire structure of a cipher/permutation and it can naturally integrate the added redundancy into the side-channel evaluation.

Concretely, the attacker builds a bipartite graph modeling the cryptographic algorithm. One set of nodes represents all the intermediate variables that are manipulated in the algorithm. The other consists of function nodes that are either leakage measurements of the intermediate variables or relations between variables. A relation between some variables x, y and z is simply a constraint that they satisfy, e.g. $x \oplus y = z$. The BP algorithm updates the initial belief on the value of each intermediate variable (derived from the leakage measurement) by using the information it possesses on the value of the other variables in conjunction with the relations between them [21].

8.1 SASCA Evaluation

We perform a SASCA-based evaluation of the Frit structure, using Belief Propagation in order to achieve the two following goals. First, we employ BP in order to concretely assess the effect of redundancy to the side-channel leakage of Frit. Second, with the aid of BP, we enable a direct comparison between the fault-detecting Frit permutation and the common fault injection (FI) countermeasure of duplication. Analytically, we run the BP algorithm twice, using two graphs. The first BP graph accounts for all the intermediate variables manipulated in the first round of the permutation, excluding the fault detection countermeasure. The second BP graph accounts for the same intermediate values, including the fault detection countermeasure, i.e. it integrates the permutation's redundancy into the SASCA attack. We simulate the leakage measurements of each 4-bit intermediate variable v using a Normal distribution $\mathcal{N}(v, \sigma^2)$, where the mean is the identity leakage function of the variable and the standard deviation σ is the same for all variables (homoscedastic assumption). The goal of the attack is to retrieve the value of the 4 least significant bits of the initial value of limb b.

Figure 12 showcases the average success rate of the different simulated attacks over 1000 experiments for $\sigma = 1$ as a function of the number of traces used for the attacks. Analyzing how fast the different success rates converge to 1, we can derive three core observations. First, we conclude from the plot that every version of BP exploits more information than a simple template attack and thus it is a potent tool for horizontal exploitation. We stress again the necessity for horizontal exploitation tools such as SASCA, due to the limited effectiveness of standard statistical templates that may result in misleading conclusions. Second we see that BP with redundancy (i.e. BP using the factor graph that includes redundancy) converges faster than BP without redundancy (i.e. BP using the factor graph that excludes redundancy). Thus we are able to observe and quantify the extra leakage penalty that is incurred by the added redundancy of the fault-detecting structure. Third, we perform a comparison between the Frit structure that uses the custom fault-detecting limb and the same Frit structure

that uses the common duplication countermeasure against FI. Note that from the point of view of a fault-injecting adversary, the countermeasures provide the same level of protection against single faults. We see that the BP attack on the FRIT structure with the extended state xconverges slower compared to the BP attack on FRIT with duplication. This shows that, the redundancy introduced by the fault-detecting structure of FRIT leaks less compared to the traditional FI countermeasure of duplication applied on the same structure. When considering SCA and FI jointly, the fault-detecting FRIT structure is improving the state of the art. With respect to fault-resistance, we have developed a countermeasure that is equivalent to duplication, however it reveals less with respect to side-channel information.

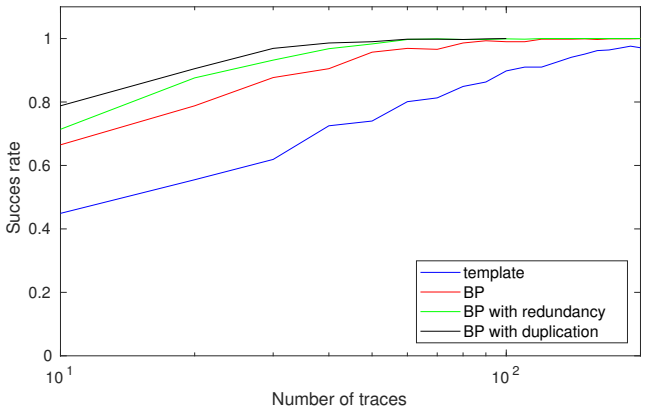


Fig. 12. Success rate of the simulated attacks

9 Conclusions and Future Work

In this paper we introduced a fault analysis countermeasure that is built-in from the design phase. The approach is based on redundancy but rather distinct from previous FA mitigations. It allows for a compact implementation in hardware i.e. on an FPGA and the performance penalties are moderate. For a software implementation on a Cortex-M4 we get an overhead of about 83% on performance that is considerably better than duplication methods and comes with verified FA resistance.

It should be noted that our implementations could be further optimized using e.g. specific features of certain platforms and this we plan to do as future work. This contribution introduced the novel fault-resistance approach and current numbers should be considered as a proof of concept.

References

1. Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., Regazzoni, F.: Midori: A block cipher for low energy. In: Advances in Cryptology – ASIACRYPT 2015. pp. 411–436. Springer Berlin Heidelberg, Berlin, Heidelberg (2015) 2
2. Banik, S., Pandey, S.K., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: Gift: A small present. In: Cryptographic Hardware and Embedded Systems – CHES 2017. pp. 321–345. Springer International Publishing (2017) 2
3. 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. pp. 7:1–7:10. WESS '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1873548.1873555 14
4. Beierle, C., Canteaut, A., Leander, G., Rotella, Y.: Proving resistance against invariant attacks: How to choose the round constants. In: CRYPTO (2). Lecture Notes in Computer Science, vol. 10402, pp. 647–678. Springer (2017) 9
5. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The skinny family of block ciphers and its low-latency variant mantis. In: Advances in Cryptology – CRYPTO 2016. pp. 123–153. Springer Berlin Heidelberg, Berlin, Heidelberg (2016) 2
6. Bernstein, D.J., Kölbl, S., Lucks, S., Massolino, P.M.C., Mendel, F., Nawaz, K., Schneider, T., Schwabe, P., Standaert, F.X., Todo, Y., et al.: Gimli: a cross-platform permutation. In: International Conference on Cryptographic Hardware and Embedded Systems. pp. 299–320. Springer (2017) 2, 8
7. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In: Cryptographic Hardware and Embedded Systems – CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer Berlin Heidelberg, Berlin, Heidelberg (2007) 2
8. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of eliminating errors in cryptographic computations. Journal of Cryptology 14(2), 101–119 (2001), http://crypto.stanford.edu/~dabo/pubs/papers/faults.ps.gz 2
9. Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçın, T.: PRINCE – a low-latency block cipher for pervasive computing applications. In: Advances in Cryptology – ASIACRYPT 2012. pp. 208–225. Springer Berlin Heidelberg, Berlin, Heidelberg (2012) 2
10. Boss, E., Grosso, V., Güneysu, T., Leander, G., Moradi, A., Schneider, T.: Strong 8-bit sboxes with efficient masking in hardware extended version. J. Cryptographic Engineering 7(2), 149–165 (2017), https://doi.org/10.1007/s13389-017-0156-7
11. Cojocar, L., Papagiannopoulos, K., Timmers, N.: Instruction duplication: Leaky and not too fault-tolerant! In: International Conference on Smart Card Research and Advanced Applications. pp. 160–179. Springer (2017) 3, 19
12. Daemen, J., Peeters, M., Assche, G.V., Rijmen, V.: Nessie proposal: Noekeon (2000), first Open Nessie Workshop. 2
13. De Cannière, C., Dunkelman, O., Knežević, M.: KATAN and KTANTAN — a family of small and efficient hardware-oriented block ciphers. In: Cryptographic Hardware and Embedded Systems - CHES 2009. pp. 272–288. Springer Berlin Heidelberg, Berlin, Heidelberg (2009) 2

14. Dobraunig, C., Eichlseder, M., Korak, T., Mangard, S., Mendel, F., Primas, R.: Exploiting ineffective fault inductions on symmetric cryptography. Tech. rep., Cryptology ePrint Archive, Report 2018/071, 2018. https://eprint. iacr. org/2018/071 (2018) 3
15. Groß, H., Mangard, S., Korak, T.: Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In: Bilgin, B., Nikova, S., Rijmen, V. (eds.) Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016. p. 3. ACM (2016), http://doi.acm.org/10.1145/2996366.2996426
16. Grosso, V., Leurent, G., Standaert, F.X., Varıcı, K.: LS-designs: Bitslice encryption for efficient masked software implementations. In: Fast Software Encryption. pp. 18–37. Springer Berlin Heidelberg, Berlin, Heidelberg (2015) 2, 3
17. Ishai, Y., Prabhakaran, M., Sahai, A., Wagner, D.: Private circuits ii: Keeping secrets in tamperable circuits. In: Vaudenay, S. (ed.) Advances in Cryptology - EUROCRYPT 2006. pp. 308–327. Springer Berlin Heidelberg, Berlin, Heidelberg (2006) 3
18. Ishai, Y., Sahai, A., Wagner, D.A.: Private circuits: Securing hardware against probing attacks. In: Boneh, D. (ed.) Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2729, pp. 463–481. Springer (2003), https://doi.org/10.1007/978-3-540-45146-4_27 25
19. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) Advances in Cryptology: Proceedings of CRYPTO'99. pp. 388–397. No. 1666 in Lecture Notes in Computer Science, Springer-Verlag (1999) 1
20. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) Advances in Cryptology – CRYPTO'96. Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer-Verlag Berlin Heidelberg (1996), http://www.cryptography.com/public/pdf/TimingAttacks.pdf 1
21. Kschischang, F.R., Frey, B.J., Loeliger, H.A.: Factor graphs and the sum-product algorithm. IEEE Transactions on information theory 47(2), 498–519 (2001) 20
22. Le Bouder, H., Lashermes, R., Linge, Y., Thomas, G., Zie, J.Y.: A multi-round side channel attack on aes using belief propagation. In: International Symposium on Foundations and Practice of Security. pp. 199–213. Springer (2016) 19
23. Mouha, N., Mennink, B., Van Herrewege, A., Watanabe, D., Preneel, B., Verbauwhede, I.: Chaskey: An efficient mac algorithm for 32-bit microcontrollers. In: Selected Areas in Cryptography – SAC 2014. pp. 306–323. Springer International Publishing (2014) 2
24. Nikova, S., Rechberger, C., Rijmen, V.: Threshold implementations against side-channel attacks and glitches. In: Ning, P., Qing, S., Li, N. (eds.) Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4307, pp. 529–545. Springer (2006), https://doi.org/10.1007/11935308_38 25
25. NIST: Federal information processing standard 202, SHA-3 standard: Permutation-based hash and extendable-output functions (August 2015), http://dx.doi.org/10.6028/NIST.FIPS.202 2
26. Regazzoni, F., Breveglieri, L., Ienne, P., Koren, I.: Interaction between fault attack countermeasures and the resistance against power analysis attacks. In: Fault Analysis in Cryptography, pp. 257–272. Springer (2012) 3, 19

27. Reparaz, O., Meyer, L.D., Bilgin, B., Arribas, V., Nikova, S., Nikov, V., Smart, N.: Capa: The spirit of beaver against physical attacks. Cryptology ePrint Archive, Report 2017/1195 (2017), https://eprint.iacr.org/2017/1195 2, 3
28. Schneider, T., Moradi, A., Güneysu, T.: ParTI – towards combined hardware countermeasures against side-channel and fault-injection attacks. In: Advances in Cryptology – CRYPTO 2016. pp. 302–332. Springer Berlin Heidelberg, Berlin, Heidelberg (2016) 2, 3
29. Research Institute for Secure Systems, N.I.o.A.I.S., Technology: Side-channel attack standard evaluation board sasebo-giii specification (2013), http://satoh.cs.uec.ac.jp/SAKURA/hardware/SAKURA-X.html 13
30. Stoffelen, K., Daemen, J.: Column parity mixers. IACR Trans. Symmetric Cryptol. 2018(1), 126–159 (2018) 2, 6
31. Toffoli, T.: Reversible computing. In: de Bakker, J.W., van Leeuwen, J. (eds.) Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings. Lecture Notes in Computer Science, vol. 85, pp. 632–644. Springer (1980), https://doi.org/10.1007/3-540-10003-2_104 6
32. Veyrat-Charvillon, N., Gérard, B., Standaert, F.X.: Soft analytical side-channel attacks. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 282–296. Springer (2014) 19

A Algebraic Properties

The round function is a degree 2 function. Moreover only 2 of the 3 branches share the same quadratic terms (a' and c'), while b' is of degree 1. If we want to estimate the degree of the permutation we can remark the following properties:

$$degree(a') \leq max(degree(c), degree(a) + degree(b)) \tag{3}$$
$$degree(c') \leq max(degree(a'), degree(b), degree(c)) \tag{4}$$
$$degree(b') = degree(a) \tag{5}$$

Equation 5 comes from the fact the only operation we apply on branch a is linear. The Equation 3 comes from the Toffoli gate, the second linear operation does not influence the degree. The Equation 4 comes from the XOR addition.

By starting with degree 1 we can upper bound the degree of each branch after one round. We define the relation of bound from the beginning of the round and the end of the round

$$bound\_degree(a') = bound\_degree(a) + bound\_degree(b) \tag{6}$$
$$bound\_degree(c') = bound\_degree(a) + bound\_degree(b) \tag{7}$$
$$bound\_degree(b') = bound\_degree(a) \tag{8}$$

It is clear that the degree of a' and c' follow the Fibonacci sequence (starting 1,2), while b' follows the Fibonacci sequence (starting 1,1). Hence to reach the maximal degree of a permutation of 382 bits we need at least 13 rounds (14 rounds such that all 3 branches reach maximal degree).

B Applying TI sharing to extended Frit

To achieve better resistance against differential power analysis (DPA) of the extended round function, one can apply a masked compiler solution as the one of Ishai, Sahai and Wagner [18]. In principle this method can provide protection against attacks of any given order, but comes at a considerable cost as it requires large amounts of randomness. Another method to protect against DPA is threshold implementation (TI) as described in [24]. It has the advantage that it can offer protection against first-order DPA, even in the presence of glitches, without the need for fresh randomness during the computation.

The non-linear steps of the round function have algebraic degree 2, so we can have a correct and non-complete TI with just 3 shares. We will share each of the 3 limbs (a, b, c) of the state in 3 shares:

$$\begin{aligned} a &= a_1 \oplus a_2 \oplus a_3 \\ b &= b_1 \oplus b_2 \oplus b_3 \\ c &= c_1 \oplus c_2 \oplus c_3 \ . \end{aligned}$$

A TI implementation of a function is provably secure against first order DPA if the sharing of its input is uniform and it each computation does not use all shares. For the Frit round function this means that $(a_1, a_2, b_1, b_2, c_1, c_2)$ must has a uniform distribution and the above equations are satisfied. For multiple iterations of the round function, the input to each round function must be uniformly shared and hence the TI sharing must preserve uniformity: one says it is *uniform.* In TI sharings of invertible mappings, uniformity is equivalent to invertibility. In other words, knowing all the shares of the function's output, one must be able to compute all the shares of the function's input. Uniformity is achieved by all invertible linear mappings λ that are implemented by applying the linear mapping to the shares independently. This is the case of all step mappings of the Frit round function, except the Toffoli gate. The Toffoli gate XORs a bitwise AND of two limbs to a third limb and finding a correct and non-complete sharing is trivial. It is easy to see that this is invertible as it leaves the the shares of the two limbs that enter the non-linear part of the computation unchanged and hence uniformity is achieved. The fact that the limbs of the extended state sum to a fixed value does not impact this as the security of TI does not rely on the (native) input to the function to have particular distributions. As a matter of fact, in general the function input can be chosen by the attacker.

As for the sharing of the extended Frit state, we will impose that for each of the shares it satisfies $d_i = a_i \oplus b_i \oplus c_i$ (assuming now for simplicity the limbs sum to 0). So after we obtained the shared input, we will compute for each of the three shares the fourth limb d_i. This copies the limb-sum redundancy to each of the shares individually. In comparison to the solution where we would share d uniformly as (d_1, d_2, d_3) with $d = d_1 \oplus d_2 \oplus d_3$ this avoids the vulnerability that inducing the same fault in two share computations would go undetected. This implies that $(a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2, c_3, d_1, d_2, d_3)$ is NOT a uniform sharing of (a, b, c, d) and it is not immediate that the extended textscFrit round function

is provably secure against first order DPA. However, we see that all FRIT step functions operate on at most 3 of the 4 limbs and the sharing of any subset of 3 or less of the 4 limbs is uniform.

So if we start from a uniform sharing $(a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2, c_3)$ of (a, b, c), then it will remain uniform during the computation. We can use this fact to re-use randomness across arbitrary sequences of FRIT computations. When we compute a TI version of the extended FRIT permutation, we require the function's (native) input (a, b, c) in three random shares that sum to the input. This can be achieved to have a random sharing of $(0, 0, 0)$ and XOR (a, b, c) to one of the three shares. For the random sharing of $(0, 0, 0)$ we need two independent random shares and compute the third random share as their XOR. However, after a FRIT computation we have the output in the form of 3 random shares summing to the output. Two of these 3 shares are random and independent. So after extracting the native input, we can reset this to a uniform sharing of $(0, 0, 0)$ by replacing the third share with the XOR of the first two. In other words, for each computation of the extended TI FRIT circuit, we can re-use the randomness available from the previous computation.

C Test Vectors

Here are four randomly generated test vectors for FRIT.

Input:
a = 0x1cf0dd8ede506b28393588a176eb82d4
b = 0xcb11d0a03097a7210a9b4f84b9af8913
c = 0x592b472995a14cd0470447b2de0a30f5
Output:
a = 0xf3f93433568ec4fe6f2d138b1cbba4c9
b = 0xfe9ea4ce979c37a487b2a8ad5e4213b6
c = 0x412bd851dc9e6621ea351a9c68f18122

Input:
a = 0xb8c5eb5d50a86ea73456ea9e3517f87a
b = 0xfc41956c73bcca8ffee303747368d630
c = 0x84ec6294709145c3df7d63430f4f98bb
Output:
a = 0x2c35190f5ebebd3503120ce4f2d325dd
b = 0x9b1f4bb17b2d708f712607208206de06
c = 0x67c016b6168544103648d3ad6e58c93e

Input:
a = 0x3acd538b0ca21867dacdb35a1f6fd142
b = 0x2082de875639e4228cfcbbe0c8eb7f33
c = 0xdc917d362a8b74b839703a7de1f4e683

Output:
a = 0xd6535b544f85e3787553f07731068da3
b = 0xfb8e049031660b8a192ebaaf5e7cf714
c = 0x1836e5e6bb4bf01b48dcb928ab9adc8f

Input:
a = 0x1342da680e728e3b93b1d03f514494dc
b = 0xab8935a2b1f684befb6d4ab8b10bf2e6
c = 0xfa6ac8b51c5e38f908c27a71dfb0f8dd
Output:
a = 0x9bb8d7b7db658c9b98a5be7c5d256d26
b = 0x307146b9767d91fd8c950a9b3bedd965
c = 0x81d55308f59bd08d8b3983406f8f93cb

D Avalanche Criterion

The following tables illustrate the average number μ of flipped bits after 7 rounds of the permutation when the bit at position *index* of the input is flipped. Those results are collected from 1000 independent inputs and the associated standard deviation σ is added in the tables.

index	μ	σ	index	μ	σ	index	μ	σ	index	μ	σ
000	192.2	9.8	032	192.3	9.6	064	192.1	9.7	096	192.3	9.6
001	191.7	10.0	033	191.8	9.5	065	191.7	9.5	097	192.2	9.6
002	191.7	10.1	034	192.2	9.5	066	191.8	9.8	098	192.6	10.0
003	192.0	9.5	035	191.8	9.7	067	192.2	10.0	099	192.1	9.5
004	191.8	9.9	036	192.0	9.9	068	191.8	9.9	100	192.2	9.9
005	192.1	10.0	037	192.3	10.0	069	192.1	10.4	101	192.4	9.5
006	192.0	9.9	038	191.9	9.6	070	191.8	9.6	102	192.1	9.7
007	191.7	9.8	039	192.0	9.7	071	191.8	9.9	103	191.6	9.7
008	191.8	9.8	040	191.7	9.8	072	192.0	10.0	104	192.1	9.9
009	192.1	9.8	041	192.2	9.8	073	191.4	9.6	105	192.1	9.4
010	191.8	9.9	042	192.1	10.2	074	192.0	9.6	106	192.1	10.2
011	192.4	9.7	043	191.8	9.9	075	191.2	9.9	107	191.2	10.0
012	191.8	9.6	044	192.2	9.5	076	191.8	9.8	108	192.1	9.6
013	191.5	10.0	045	191.5	9.9	077	192.0	9.7	109	191.6	10.3
014	192.0	9.6	046	192.4	9.7	078	192.1	10.0	110	192.2	9.8
015	191.9	9.7	047	192.1	9.7	079	191.6	9.8	111	192.0	9.9
016	191.7	9.9	048	191.5	9.8	080	192.6	9.9	112	191.9	9.7
017	192.1	9.7	049	192.2	9.6	081	192.0	9.9	113	191.8	9.7
018	192.2	9.7	050	191.9	9.8	082	191.9	10.0	114	192.3	9.9
019	191.4	9.9	051	191.9	9.4	083	192.0	10.1	115	191.7	10.3
020	191.8	9.7	052	192.0	9.5	084	191.8	9.8	116	192.5	9.6
021	191.9	9.4	053	192.0	9.9	085	192.3	10.0	117	192.0	9.6
022	192.2	10.1	054	192.1	10.1	086	191.9	9.5	118	192.1	9.5
023	192.1	9.8	055	191.9	9.9	087	192.0	9.6	119	192.1	9.8
024	191.6	9.6	056	191.9	10.4	088	192.2	9.9	120	192.1	9.8
025	192.0	9.9	057	192.5	9.9	089	192.0	9.8	121	191.8	9.7
026	191.9	9.5	058	191.9	9.7	090	191.9	9.9	122	192.1	9.5
027	192.3	9.5	059	192.0	10.0	091	192.2	9.7	123	191.7	9.9
028	192.2	9.9	060	191.7	9.7	092	191.9	9.8	124	192.0	9.9
029	192.0	9.8	061	191.8	9.8	093	192.5	9.7	125	192.1	9.8
030	192.3	9.9	062	191.5	10.0	094	192.3	9.5	126	192.1	9.9
031	191.4	9.8	063	191.5	9.4	095	192.0	10.0	127	192.5	9.8

index	μ	σ	index	μ	σ	index	μ	σ	index	μ	σ
128	191.8	10.0	160	192.2	9.9	192	191.7	9.6	224	191.5	9.9
129	191.8	9.8	161	191.6	10.2	193	192.2	10.0	225	192.5	10.1
130	191.7	9.7	162	191.4	9.7	194	192.1	10.0	226	191.5	9.6
131	191.7	10.2	163	191.7	10.0	195	192.1	10.0	227	191.9	10.0
132	192.0	9.8	164	191.6	9.9	196	191.9	10.0	228	192.0	9.9
133	191.5	10.0	165	191.8	9.7	197	191.8	9.6	229	191.4	9.8
134	191.6	9.7	166	191.9	9.5	198	192.4	10.0	230	191.8	9.6
135	192.1	9.8	167	192.1	10.4	199	192.1	9.9	231	192.0	9.8
136	191.8	9.9	168	191.9	10.1	200	192.4	9.9	232	192.2	9.5
137	191.9	9.8	169	191.9	9.9	201	191.6	9.8	233	192.4	10.1
138	191.9	10.0	170	191.8	9.8	202	191.9	10.2	234	191.8	9.9
139	191.6	10.0	171	191.9	9.9	203	192.2	9.7	235	191.7	9.7
140	192.1	10.0	172	191.7	10.1	204	191.9	9.8	236	192.0	9.8
141	191.9	10.0	173	191.8	9.9	205	192.1	9.7	237	191.6	9.5
142	192.2	9.4	174	191.7	9.8	206	191.5	10.0	238	191.9	9.7
143	192.4	9.6	175	192.1	9.9	207	192.1	9.7	239	191.9	9.7
144	191.6	10.2	176	192.2	9.6	208	191.9	10.0	240	192.1	10.0
145	191.8	10.0	177	192.0	9.7	209	191.8	10.0	241	192.0	10.3
146	191.8	9.7	178	191.4	9.7	210	192.0	9.6	242	191.9	9.6
147	191.9	9.6	179	192.5	9.9	211	192.0	9.6	243	191.9	9.8
148	191.7	10.2	180	191.9	10.0	212	191.6	9.6	244	192.1	9.6
149	191.7	9.7	181	192.3	9.8	213	192.1	9.9	245	191.9	9.9
150	192.2	10.2	182	192.0	9.6	214	191.6	9.5	246	192.7	9.9
151	192.0	9.7	183	191.6	10.1	215	191.7	10.0	247	191.8	9.2
152	192.1	9.9	184	191.8	10.0	216	191.8	9.8	248	192.1	9.9
153	191.8	9.5	185	191.5	10.0	217	192.0	10.1	249	191.3	10.0
154	192.3	10.0	186	191.7	9.8	218	191.9	9.9	250	191.8	9.5
155	191.5	9.4	187	192.4	9.7	219	192.1	9.5	251	192.2	9.9
156	191.9	9.4	188	192.1	9.7	220	191.9	9.5	252	191.7	9.6
157	191.8	10.1	189	192.0	9.9	221	191.6	9.9	253	191.9	10.0
158	192.0	9.7	190	191.6	9.8	222	191.9	9.6	254	191.8	9.4
159	191.7	9.8	191	191.4	9.8	223	191.5	9.8	255	191.5	10.0

index	μ	σ	index	μ	σ	index	μ	σ	index	μ	σ
256	191.4	9.7	288	191.5	9.4	320	192.1	10.0	352	191.9	9.9
257	191.7	10.1	289	192.0	9.9	321	192.0	9.7	353	191.9	10.1
258	191.8	9.7	290	191.8	9.9	322	191.9	9.9	354	191.8	9.8
259	191.7	9.6	291	192.1	9.9	323	192.2	9.8	355	191.9	9.5
260	191.9	10.1	292	192.0	9.8	324	191.2	9.4	356	192.2	10.2
261	192.0	9.8	293	191.7	9.8	325	192.0	9.7	357	191.5	9.9
262	192.2	9.3	294	192.3	9.5	326	191.9	9.7	358	192.5	10.0
263	191.8	9.8	295	191.8	9.6	327	192.2	9.9	359	191.4	10.2
264	191.8	9.9	296	192.0	9.6	328	191.6	9.8	360	192.4	10.1
265	192.5	9.6	297	192.4	9.8	329	191.5	9.8	361	191.9	9.4
266	191.7	9.4	298	192.2	9.8	330	191.9	10.3	362	191.8	9.6
267	192.3	10.0	299	192.3	9.9	331	191.4	9.9	363	192.1	9.7
268	191.7	10.0	300	191.8	9.7	332	191.3	9.7	364	192.0	9.7
269	191.6	9.7	301	191.6	10.1	333	191.6	9.7	365	191.9	9.9
270	192.0	9.6	302	191.8	10.1	334	192.1	10.2	366	192.4	9.9
271	192.5	10.0	303	192.1	9.4	335	192.0	10.0	367	191.6	9.5
272	191.7	9.5	304	191.8	9.7	336	191.6	10.1	368	191.7	10.1
273	191.6	9.7	305	192.2	9.7	337	192.0	9.7	369	192.1	10.0
274	191.4	9.7	306	191.7	9.6	338	191.9	9.7	370	191.6	9.2
275	191.4	9.9	307	192.2	9.5	339	191.8	10.0	371	191.1	10.2
276	191.4	9.9	308	191.8	9.6	340	191.8	9.7	372	191.4	9.7
277	192.0	10.4	309	191.8	9.4	341	191.8	10.0	373	191.8	9.8
278	191.6	10.0	310	191.4	9.4	342	192.3	9.3	374	192.2	9.6
279	192.3	9.7	311	191.8	9.9	343	192.2	9.6	375	192.3	9.5
280	191.8	9.5	312	191.5	9.6	344	191.9	9.8	376	191.9	9.6
281	192.0	9.4	313	192.4	9.9	345	192.6	10.2	377	191.8	10.0
282	191.9	9.7	314	192.3	9.6	346	192.0	9.9	378	191.8	10.0
283	192.4	9.8	315	192.5	9.4	347	191.7	9.9	379	191.8	9.8
284	192.2	9.9	316	192.0	9.6	348	191.9	9.4	380	192.2	9.6
285	192.1	9.7	317	191.6	9.9	349	191.7	9.5	381	191.8	9.2
286	191.3	9.9	318	191.9	9.9	350	192.1	10.0	382	192.0	9.5
287	192.4	9.9	319	192.0	9.8	351	191.9	9.7	383	192.2	10.1