Arithmetic Logic Units with High Error Detection Rates to Counteract Fault Attacks

Marcel Medwed
IAIK, Graz University of Technology, Austria
Crypto Group, Université catholique de Louvain, Belgium
Email: Marcel.Medwed@uclouvain.be

Stefan Mangard
Infineon Technologies AG
Munich, Germany
Email: Stefan.Mangard@infineon.com

Abstract—Modern security-aware embedded systems need protection against fault attacks. These attacks rely on intentionally induced faults. Such intentional faults have not only a different origin, but also a different nature than errors that fault-tolerant systems usually have to face. For instance an adversary who attacks the circuit with two lasers can potentially induce two errors at different positions. Such errors can not only defeat simple double modular redundancy schemes, but as we show, also naive schemes based on any linear code over GF(2). In this article, we describe arithmetic logic units (ALUs) which provide high error detection rates even in the presence of such errors. The contribution in this article is threefold. First, we show that the minimum weight of an undetected error is no longer defined by the code distance when certain arithmetic and logic operations are applied to the codewords. As a result, additional hardware is needed to preserve the minimum error weight for a given code. Second, we show that for multi-residue codes, these delicate operations are rare in typical smart card applications. This allows for an efficient time-area trade-off for checking the codewords and thus to significantly reduce the hardware costs for such a protected ALU. Third, we implement the proposed architectures and study the influence of the register file and a multiplier on the area and on the critical path.

I. INTRODUCTION

In more and more applications, cryptographic operations are performed on embedded processors. Some of the most important applications in this context are payment, identification, access control, digital rights management and IP protection. In order to guarantee the security of these applications, it is necessary to implement countermeasures against physical attacks on the embedded processors.

During the last 15 years numerous physical attacks have been published that allow the extraction of secret information based on the observation or manipulation of an embedded device and its environment. Typical examples of physical attacks are timing attacks [8], power analysis attacks [9] and fault attacks [2], [3]. While timing and power analysis attacks have received much attention already immediately after their publication, not so much attention has been paid on fault attacks so far.

However, fault attacks become increasingly important. Meanwhile there exist several publication that discuss methods to induce faults in order to reveal secret information [4], [15]–[18]. In general there are two types of fault attacks. On the one hand, there are global fault attacks on the systems. In this case, the attacker has only limited control over the type and location of the fault. On the other hand, there are local fault attacks. Using sophisticated laser equipment it is possible to bypass global countermeasures such as light sensors.

In order to build a secure embedded processor, it is necessary to provide error detection mechanisms that can detect global attacks as wells as very local attacks. A solution to achieve this goal is to duplicate the design and to hence build a dual modular redundant (DMR) system. This approach allows high error detection rates, but is also associated with significant costs. Furthermore, simple DMR schemes can potentially be defeated by so-called second-order fault attacks [7]. In this article, we present and analyze a novel code-protected ALU that provides a high error detection rate on the one hand and that has less area requirements than a DMR ALU. In fact, our ALU preserves the code distance throughout all operations and has an error detection rate of 99,998%. Despite this, we save 24.8% of area compared to a DMR ALU.

A. Related Work

Coding schemes for processors are a research topic since decades. Already in 1970, Rao published a paper on how to use bi-residue codes for error correction for arithmetic operations [12]. Later on, residue prediction formulas for all common arithmetic and logic operations have been developed and also deployed in computer architectures [13], [14]. Elliot and Sayers on the other hand equip an ALU with Hamming code predictors to increase its reliability [5].

All above works only consider single-bit and in some cases double-bit faults. Furthermore, double-bit faults are not considered to appear in both operands of a calculation simultaneously since this is unrealistic if the faults occur unintentionally. As a result, those architectures are not suited to protect against fault attacks. The problem of using error-detection codes to protect complete ALU/register-file units against fault attacks did not receive much attention so far.

Protecting dedicated cryptographic hardware against fault attacks on the other hand is indeed a vivid research topic. An overview of protection mechanisms for AES implementations can be found in [10]. The work presented in [6] protects polynomial multipliers using parity schemes. However, although the protection of sensitive parts such as cryptographic co-processors is necessary, it is not sufficient. Already before fault attacks were presented in the cryptographic community, they

have been used to attack pay-TV cards [1]. Here, the operating system was the point of attack. Due to manipulated counter or address values, sensitive memory regions were sent by the cards to their output.

B. Our Contribution

As mentioned above, fault-attack protected systems must be resistant to global attacks. Such attacks result in a random error and thus a high redundancy suffices to repel global attacks. A typical amount of redundancy would be 16 bits which accounts for a detection rate of $1 - 2^{16} \approx 99.998\%$. However, there are also local attacks. So far no work has shown the effect of precise low weight errors which affect both operands involved in an logic or arithmetic operation. To grasp this effect we define the *operational distance* as a metric which states the minimum weight for an undetected error for a given code under a given operation. In fact, we show that the operational distance of an arbitrary linear code over GF(2) is two and the one of an arbitrary multi-residue code is three for some operations.

Based on these results, we define a general architecture for which the minimum weight of an undetected error is equal to the code distance rather than to the operational distance. In terms of security this architecture is superior to existing approaches. However, especially for multi-residue codes, the hardware costs of such an architecture need to be reduced. To do so, we use the fact that only some operations have an operational distance smaller than the code distance. By analyzing the instruction frequencies for typical smart card applications we can further observe that those operations are rare for multi-residue codes. Based on these observations, we design a new architecture which reduces the hardware overhead for a multi-residue code based ALU by 90% points.

Finally, we synthesize these architectures for a $0.35\mu m$ CMOS standard cell library and compare the area costs and the critical path delays to the ones of a DMR-based approach. For further analyses, we also take the effect of the register file and the one of a possible multiplier into account. To the best of our knowledge this is the first work to estimate the costs of protecting a general purpose datapath against fault attacks[1].

The remainder of the paper is organized as follows: Section II discusses linear and multi-residue codes and analyzes their operational distances. An architecture for which the minimum error weight is equal to the code distance is presented in the first part of Section III. In the second part of this section, we analyze instruction frequencies for smart card applications and discuss optimizations for the previously presented architecture. In Section IV, the different hardware designs are discussed and compared. Finally, conclusions are drawn in Section V.

II. SUITABLE CODES

This section discusses binary linear codes and multi-residue codes. We look at their basic properties and how they can be

---

[1]Note that we are only concerned with data integrity and that address-bus manipulations and attacks on the program flow (glitches, power spikes) have to be handled separately.

used to protect arithmetic and logic operations. Eventually, we investigate their operational distance for common ALU operations. For this purpose we use the following definition.

Definition Let $\mathcal{C}$ be a channel code and f a binary operation $f(a, b) = c$ with $a, b, c \in \mathcal{C}$. Furthermore, let $e = (e_a, e_b)$ be an error which affects both operands, such that ($\tilde{a} = a + e_a, \tilde{b} = b + e_b$). We define the Hamming weight of e as $W_H(e) = W_H(e_a) + W_H(e_b)$. For an undetected error, we demand $c \neq \tilde{c} = f(\tilde{a}, \tilde{b})$ but $\tilde{c} \in \mathcal{C}$. We define the **operational distance of $\mathcal{C}$ regarding f** as the minimum Hamming weight $W_H(e)$ needed to produce an undetected error.

A. Linear Codes

A $[n, k]$ linear code encodes a k-bit dataword into an n-bit codeword. This is done by multiplying the dataword $d \in \mathbb{F}_q^k$ with the $k \times n$ generator matrix $\mathbf{G}$. We only look at the case where $q = 2$. In systematic form, $\mathbf{G}$ can be written as $[\mathbf{I}|\mathbf{P}]$ where $\mathbf{I}$ is a $k \times k$ identity matrix and $\mathbf{P}$ is the $k \times r$ parity matrix. A codeword is error free, if its product with the $r \times n$ parity check matrix $\mathbf{H}$ is the zero vector, where $\mathbf{H} = [\mathbf{P}^T|\mathbf{I}]$.

The code distance D is the minimum pairwise Hamming distance between all codewords. Consequently, every error with Hamming weight smaller D is detected with certainty. For the parameters $[n = 48, k = 32]$, the best known binary linear code has distance 6.

In the following we recall the parity generation for arithmetic and logic operations on linear codes. We denote datawords by $d_a, d_b, \ldots$ and the corresponding parities by $p_a = d_a\mathbf{P}, p_b = d_b\mathbf{P}, \ldots$. The logic-and, logic-or, exclusive-or and integer addition operations are denoted by $\wedge, \vee, \oplus$ and $+$ respectively.

For $d_c = d_a \oplus d_b$ the corresponding parity is $p_c = p_a \oplus p_b$. The exclusive-or operation is the same as addition over $\mathbb{F}_2$ which is the base field of the code. Hence, this operation is natively supported by the code. As a consequence, neither auxiliary values are needed to calculate the correct parity nor is the operational distance smaller than the code distance. This means that even if both operands are affected by errors, the Hamming weight of the errors' sum has to be at least D. Otherwise the error is detected with certainty.

Shift operations and other unary linear operations involve only one operand. Thus they preserve the distance of the code as well. For linear codes the result of an unary operation $d_c = u(d_a)$ has the corresponding parity $p_c = p_a \oplus (d_a \oplus u(d_a))\mathbf{P}$.

The logic-and and the logic-or operation need auxiliary values to generate a valid parity. These are parities which are derived from intermediate values during the calculation. For $d_c = d_a \wedge d_b$, the parity is given by $p_c = p_a \oplus p_b \oplus (d_a \vee d_b)\mathbf{P}$. Similarly, for $d_c = d_a \vee d_b$, the parity is given by $p_c = p_a \oplus p_b \oplus (d_a \wedge d_b)\mathbf{P}$.

To handle integer addition, the carry bits of the arithmetic operation have to be incorporated in the parity of the sum. We denote the carry bits, generated during an addition, by d_{carry}, where the least significant bit is the carry-in. The carry-out bit is not part of d_{carry}, thus d_{carry} is also k bits long. The parity

calculation for addition follows the observation that $d_s = d_a + d_b = d_a \oplus d_b \oplus d_{carry}$. Therefore, $p_s = p_a \oplus p_b \oplus d_{carry}\mathbf{P}$. If the carry generation itself is corrupted, the error adds to the sum as well as to the parity. As a consequence, any error in the carry generation stays undetected.

For exclusive-or and unary linear operations the operational distance equals the code distance. The operational distance for the other three operations is stated in the following:

Theorem 1. *The operational distance for every binary linear code regarding logic-and, logic-or and integer addition is two.*

Proof: We can write the three operations as $\tilde{c} = f(\tilde{a}, \tilde{b})$ with the parity written as $\tilde{p}_c = p_a \oplus p_b \oplus g(\tilde{d}_a, \tilde{d}_b)\mathbf{P}$. The encoded result of g already depends on the erroneous operands, thus this part of the parity already depends on $\tilde{d}_c$. The only portions which still depend on the original parities are p_a and p_b. Hence, an error stays undetected, if it does not affect the sum $p_a \oplus p_b$. This is the case for errors (e_a, e_b) for which $e_a = e_b$. The smallest error e_a has $W_H = 1$, thus $W_H(e)$ is at least two. ∎

Linear codes are well studied and can achieve a high distance. However, for non-native operations like logic-and, logic-or and integer addition, the operational distance is only two. Furthermore, for integer addition, any corruption of the carry vector stays undetected. Additionally, linear codes do not support multiplication, because they are not defined for operands larger than k bits.

B. Multi-Residue Codes

Multi-residue codes are arithmetic codes, defined by a set of moduli $\mathbf{m} = \{m_1, \ldots, m_l\}$. The check symbol of a dataword a is given by $\mathbf{p_a} = \{d_a \mod m_1, \ldots, d_a \mod m_l\}$. Finding a suitable set of moduli with a high distance needs to be done via exhaustive search. Following the conditions given in [11], a set which assures $D = 4$ has been found for $r = 16$. In particular, the check base $\mathbf{m}$ was chosen as $\{5, 7, 17, 31\}$.

The native operation of these codes is the addition. It can be verified that indeed, for $d_c = d_a + d_b$, the corresponding parity $\mathbf{p_c} = \mathbf{p_a} + \mathbf{p_b} \mod \mathbf{m}$. Here, addition and reduction are applied componentwise to the elements of the parity vector. Furthermore, also multiplication can be performed without auxiliary values, given by $d_c = d_a \cdot d_b$ and $\mathbf{p_c} = \mathbf{p_a} \cdot \mathbf{p_b} \mod \mathbf{m}$. Analog to the exclusive-or for linear codes, addition preserves the distance for multi-residue codes.

For Boolean operations, auxiliary values are needed. The used relation is $d_c = d_a + d_b = 2 \cdot (d_a \wedge d_b) + (d_a \oplus d_b)$. From that it follows, that the parity for $d_c = d_a \wedge d_b$ is calculated by $\mathbf{p_c} = (\mathbf{p_a} + \mathbf{p_b} - (d_a \oplus d_b \mod \mathbf{m}))/2$. Analog, for $d_c = d_a \vee d_b$, $\mathbf{p_c} = (\mathbf{p_a} + \mathbf{p_b} + (d_a \oplus d_b \mod \mathbf{m}))/2$ and eventually, for $d_c = d_a \oplus d_b$, $\mathbf{p_c} = \mathbf{p_a} + \mathbf{p_b} - (2 \cdot (d_a \wedge d_b) \mod \mathbf{m})$. It is not too surprising that also for these operations, the code distance drops if both operands are affected by errors.

Theorem 2. *The operational distance for every multi-residue code regarding logic-and, logic-or and exclusive-or is three.*

Proof: We can write the three operations as $\tilde{c} = f(\tilde{a}, \tilde{b})$

with the parity written as $\tilde{\mathbf{p}}_c = \mathbf{p_a} + \mathbf{p_b} + (g(\tilde{d}_a, \tilde{d}_b) \mod \mathbf{m})$. The encoded result of g already depends on the erroneous operands, thus this part of the parity already depends on $\tilde{d}_c$. The only portions which still depend on the original parities are $\mathbf{p_a}$ and $\mathbf{p_b}$. Hence, an error stays undetected, if it does not affect the sum $\mathbf{p_a} + \mathbf{p_b}$. This is given if $\tilde{a} + \tilde{b} = a + b$. As a consequence, for an error $e = (e_a, e_b)$, $e_a = -e_b$ must hold. The smallest such e is $(1, -1)$, where the arithmetic error -1 can be realized with 2 bits: $2_d - 1_d = 10_b \oplus 11_b$. Thus, $W_H(e) = 3$. ∎

Summarizing, multi-residue codes have several advantages over binary linear codes. First, the smallest operational distance for multi-residue codes is three, opposed to two for linear codes. Second, multi-residue codes are prone to arithmetic errors, which are in turn data dependent. Thus, it is harder to induce precise errors. Third, they do not have direct weaknesses like the carry corruption problem for linear codes and finally they natively support multiplication.

III. GENERAL AND OPTIMIZED ARCHITECTURES

In the previous section it became clear that non-native operations need special treatment for two reasons. First, they need auxiliary values to predict the correct check symbol. Second, the use of these auxiliary values decreases the operational distance. In this section we look for the minimum effort needed to preserve the code distance and give a general architecture. Furthermore, we discuss a time-area trade-off optimization of this architecture for multi-residue codes.
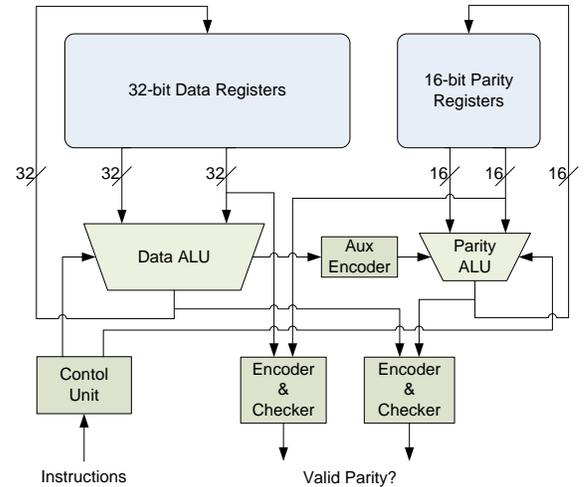

Fig. 1. General architecture of the protected datapath.

A. Distance-preserving Measures

All investigated codes preserve the code distance if only one operand is affected, even for non-native operations. Problems only arise if $e_a \neq 0$ and $e_b \neq 0$. In this case, it is possible that an operation produces a valid but incorrect result although $W_H(e) < D$. That is, the total Hamming weight of the error is smaller than the code distance.

However, this demands $W_H(e_a) < D$ and $W_H(e_b) < D$. As a consequence, e_a as well as e_b is detectable with certainty,
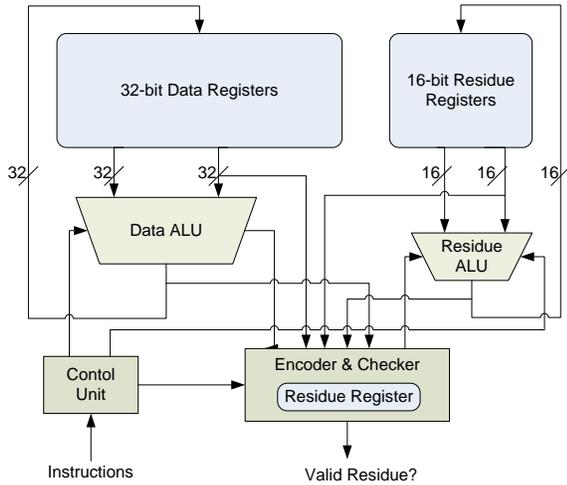
Fig. 2. Optimized multi-residue code ALU.

Instr.	AES	Boot	Shell	ECC
Arith.	17,62%	23,83%	13,23%	17,46%
Branch	0,33%	28,42%	25,62%	4,66%
Cmp	0,22%	11,72%	16,37%	2,22%
Load/Store	40,72%	23,60%	23,02%	54,73%
Logic	19,53%	3,57%	6,93%	0,70%
Misc	0,07%	1,20%	0,68%	1,46%
Mov	21,51%	7,42%	14,01%	15,48%
Mul	0,00%	0,24%	0,13%	3,30%

if we check $\tilde{a}$ and $\tilde{b}$. Furthermore, since such errors affect both operands, it is sufficient to check only one operand. It follows, that checking the result and in addition one of the two operands preserves the distance. For an implementation this means that three encoders and two checkers are needed. A general architecture which fulfills this requirement is depicted in Figure 1.

Depending on the code, parity generation is a complex task and two additional encoders are expensive. On the other hand, native operations only need one encoder for checking the result. This raises the interesting question of how often non-native operations have to be performed. Assuming that they are rare, it would be possible to save area by spending two additional clock cycles and re-using the encoder.

B. Instruction Frequency Analysis

In order to determine the need of additional encodings we analyzed the instruction stream of an ARM7TDMI-S microprocessor. For that purpose we modified the ARM emulator *Skyeye* to obtain the instruction streams for various application. The instruction stream was then parsed for instructions. From this information an instruction profile for four different scenarios, of which we believe that they are representative for smart cards, was created.

The first code we evaluated was an optimized 32-bit implementation of the Advanced Encryption Standard (AES). The second code performed an elliptic curve scalar multiplication on the standardized NIST curve P-192. Eventually, we used μC-Linux as a candidate for an operating system. In the last case we did the profiling for the boot process as well as for basic shell activities.

The outcome of the instruction frequency analysis can be seen in Table I. For all four scenarios, data transfer operations (Load/Store, Mov) together with arithmetic operations (Arith., Cmp) account for more than 50% of the instructions. A good deal of the arithmetic instructions is used for address arithmetic. An AES encryption needs 20% of logic operations,

but for the remaining scenarios, they are rather rare. In fact, during normal operation, the share of logic operations is at maximum 6,93%.

For linear codes, this has minor consequences. Their only native operation is the exclusive-or and hence using three encoders is inevitable. However, for multi-residue codes, the share of non-native operations is roughly 7%. Thus, using only one encoder, but three cycles for every non-native operation would cause a time overhead of about 14%. The resulting savings in area are discussed in Section IV.

IV. HARDWARE RESULTS

Taking the observations and results from above into account, we synthesized two code-protected ALUs and one DMR ALU using Cadence's RTL Compiler 8.1 and the AMS C35b4 standard cell library. The first design incorporates linear codes according to Figure 1. The second one implements the optimized multi-residue code based datapath according to Figure 2. The DMR ALU consists of two standard ALUs, two register files and one comparator.

A. Area of the Combinatorial Part

In Table II, it can be seen that a DMR ALU occupies 1763 gate equivalents (GE). The linear code ALU is 41% larger when incorporating three encoders. The multi-residue code ALU on the other hand is around 74% larger than the DMR ALU. A multi-residue code ALU with three encoders would be even 164% larger. However, by adding 14% time redundancy and thus removing two encoders from the design, it was possible to reduce the area overhead by 90% points. This shows that code-protected ALUs with large redundancy introduce a significant overhead in the combinatorial part of the circuit compared to a DMR ALU.

At first glance, it looks peculiar that the standard ALU for the linear-code design is much larger. The reason for this is that we needed to access the internal carry vector of the adder. Therefore, we had to implement this adder as an array of full adders instead of using the optimized instance of the standard cell library. As a result, this ALU is 68% larger than the others. Considering that most structures within the multi-residue encoders and the multi-residue ALU are also unoptimized full-adder arrays, it might be possible to also decrease their area significantly by having optimized carry-save adders.

DRM ALU		Residue ALU		Linear ALU	
1. ALU	641	ALU	649	ALU	1080
2. ALU	641	Res ALU	721	Lin ALU	110
Controller	481	Encoder	791	1. Encoder	283
		Checker + Reg.	301	2. Encoder	283
		Controller	611	3. Encoder	283
				Controller	447
	1763		**3073**		**2486**

# Registers	Absolute memory savings	Relative savings multi-res. codes	Relative savings linear codes
4	580 GE	-18%	-4%
8	1160 GE	-2%	7%
16	2320 GE	9%	14%
32	4640 GE	16%	19%

B. Total Area

By just looking at the combinatorial part of the circuit, implementing an ALU using coding techniques does not seem to be very attractive. However, looking at the whole datapath, that is ALU plus register file, it has an important advantage. Using coding techniques, the amount of redundancy can be adapted to the application's needs. For our design decisions, we assumed that a redundancy of 16 bits is sufficient to counteract global fault attacks. Compared to DMR approaches this means that the register file becomes 25% smaller. Since registers occupy a significant part of the area, also these savings are significant. Looking at an architecture which features 32 registers, the register file of the code based approach is 4640 GE smaller (see Table III)[2]. In general, a code based approach pays off from 5 registers upwards for linear codes and from 9 registers upwards for multi-residue codes. Furthermore, the savings of both approaches converge towards the 25% percent limit as the number of registers increases.

C. Timing Behavior

Besides the required area, another interesting property of a hardware implementation is the timing behavior. Also the comparison of the timing delay is difficult due to the same optimization reasons as stated in Section IV-A. The DMR based ALU showed a delay of 19.6ns in the critical path. The linear code based ALU had a critical path of 31.8ns. Within the critical path for the linear code based design lies the adder and the encoder. This is because the encoder has to wait for the carry vector during an addition. Out of these 31.8ns, the encoder and the parity ALU account only for 3.8ns. Again it should be noted that the full adder array introduces a higher delay than the adder instantiated by the synthesizer. In fact the

---

[2]As the actual realization of a register file is very context specific, we settled for a basic flip-flop based implementation.

---

delay difference is about 10ns. The critical path of the multi-residue code design is 28.7ns long. Here, the adder and the encoder lie within this path but not the residue ALUs as they operate independently for arithmetic operations. The encoder itself introduces a delay of 10ns.

D. Adding a Multiplier

Until now, we did not consider the protection of a multiplier. However, as some 32-bit smart cards already feature a multiplier (e.g. the ST33 family is based on the ARM Cortex-M3 platform) and further multi-residue codes support multiplication natively, adding such instructions seems to be very appealing. Additionally, the complexity of multiplying the residues is much smaller than for the actual data itself, thus we expect significant area savings.

Basically, we can use the fact that the product of the data part is congruent to the product of the residues. However, this only holds for the full 64-bit result. Within the microcontroller, we need to store the result as two 32-bit words and thus we have to derive their residues before storing these values. In the following we use the $hi(\cdot)$ function to get the upper 32 bits of the 64-bit value and $lo(\cdot)$ to get the lower half of the product. We can observe that $lo(c) = c - hi(c) \cdot 2^{32}$ and $hi(c) = (c - lo(c)) \cdot 2^{-32}$. Analog to that, we can calculate the residues with $\mathbf{p_{lo(c)}} = \mathbf{p_c} - (hi(c) \mod \mathbf{m}) \cdot (2^{32} \mod \mathbf{m})$ and $\mathbf{p_{hi(c)}} = (\mathbf{p_c} - (lo(c) \mod \mathbf{m})) \cdot (2^{-32} \mod \mathbf{m})$.

The calculation of the auxiliary values $(hi(c) \mod \mathbf{m})$ and $(lo(c) \mod \mathbf{m})$ is inevitable. Fortunately, we can spare the calculation of $(2^{32} \mod \mathbf{m})$ and $(2^{-32} \mod \mathbf{m})$. This is because we chose the base $\mathbf{m}$ in a way such that the multiplications and divisions can either be spared completely or expressed by constant rotations which just need rewiring.

The multiplication itself needs two cycles as we have to write back each word of the result separately. During each cycle we have to perform an encoding and a subtraction in order to get the residue-vector. Additionally, we need to check the encoded result itself against the calculated residues. In order to perform everything within two cycles and to keep the critical path as short as possible, we use the following strategy:

In the first cycle, the product c is computed and the upper 32 bits of the result ($hi(c)$) are written back. For the residue calculation we need to encode the lower 32 bits ($lo(c)$). These bits are ready before the upper ones. As a result, we can interleave the encoding with the multiplication. Of course, the encoding and the subtraction still take longer than the carry propagation inside the multiplier, but we save the delay of a 32-bit carry propagation. By now we only encoded the lower 32 bits, thus we cannot check the upper bits before writing them back. Instead, we store the values $\mathbf{p_{hi(c)}}$ and $(lo(c) \mod \mathbf{m})$ in separate check registers.

In the second cycle we encode the upper bits of the product and subtract the result times 2^{32} from $\mathbf{p_c}$ in order to get the residues for the lower bits. In this case we can directly access the product without delay, thus the second cycle does not lie in the critical path. In addition to writing back the lower bits

and their residue-vector, we store the values $\mathbf{p_{lo(c)}}$ and $(hi(c) \bmod \mathbf{m})$ in separate check registers.

Until now, we only calculated the residues and wrote them back to the register file but did not check the result. This is done via the four separate check registers. They are permanently connected to a comparator which checks if the two *hi*-registers and the two *lo*-registers hold the same value. The result of this comparison is ANDed with the global *ok_s* signal, except for the two multiplication cycles where the comparison would fail.

E. Area and Timing with Multiplier

The DMR based ALU without register file and with two 32-bit multipliers occupies 13134 GE. The multi-residue ALU on the other hand occupies only 9950 GE. This means that our code based approach is 24.2% smaller.

The critical path of the DMR based approach is 32.2ns long. The multi-residue ALU takes 35.1ns to compute the result. The encoding and the subtraction itself take 15.2ns, but by interleaving the multiplication and the residue calculation the critical path increases only by 2.9ns instead of 15.2ns. In percent this means that the critical path of the multi-residue approach is only 9% longer than in the DMR design.

V. CONCLUSION

In this article we discussed code protected datapaths as a more robust alternative to dual modular redundant designs. In terms of security, the code protected designs are superior to the DMR based approach because they detect errors of small multiplicity with certainty. In the case of the linear ALU all errors up to a multiplicity of five are detected. The multi-residue ALU on the other hand detects all errors up to a multiplicity of three with certainty. However, it should be noted that it is data dependent whether the desired error can even be injected in the case of multi-residue codes. Within the code protected designs, the multi-residue approach is superior for two reasons. First, it has no direct weak points, like the linear ALU has with the carry generation, where theoretically a single-bit fault would succeed, unless further measures are taken. Second, it allows to incorporate a multiplier.

In terms of area, both basic ALU designs are larger than a DMR approach. However, when also considering the area of the register file, both code protected designs become smaller than the DMR design. Additionally, the synthesizer showed to be very good at optimizing standard structures, but did not recognize for instance carry-save adders or full-adder arrays as such. Therefore, there should be potential to further optimize the multi-residue design as a great deal of its area is occupied by such structures.

In terms of performance, both code based designs are slower than the DMR approach as the result of the standard ALU is always fed into an encoder and sometimes further into the parity/residue ALU. That is, the critical path increases by 62% for the linear ALU and by 46% for the multi-residue ALU.

However, as soon as we introduce a 32-bit multiplier in the design, things change in favor of the multi-residue ALU. That

is, since the residue multiplications are much less complex than the full 32-bit multiplication, the multi-residue ALU now becomes 24.8% smaller than the DMR ALU. Furthermore, it is now possible to interleave the multiplication and the encoding and thus decrease the timing overhead to 9%.

We conclude that the multi-residue ALU is superior in terms of robustness and provides the best security/area tradeoff for the standard ALU. When more complex operations like multiplications are implemented it is definitely the best choice.

REFERENCES

[1] R. J. Anderson and M. G. Kuhn, "Tamper Resistance - a Cautionary Note," in *Proceedings of the 2nd USENIX Workshop on Electronic Commerce, Oakland, California, November 18-21, 1996.* USENIX Association, November 1996, pp. 1–11, ISBN 1-880446-83-9.

[2] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," in *Advances in Cryptology*, CRYPTO '97, LNCS 1294. Springer, 1997, pp. 513–525.

[3] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)," in *Advances in Cryptology*, EUROCRYPT '97, LNCS 1233. Springer, 1997, pp. 37–51.

[4] M. Boreale, "Attacking Right-to-Left Modular Exponentiation with Timely Random Faults," in *Fault Diagnosis and Tolerance in Cryptography*, FDTC 2006, LNCS 4236. Springer, October 2006, pp. 24–35.

[5] I. Elliott and I. Sayers, "Implementation of 32-bit RISC processor incorporating hardware concurrent error detection and correction," in *Computers and Digital Techniques, IEE Proceedings E*, vol. 137, Jan 1990, pp. 88–102.

[6] S. T. J. Fenn, M. Gössel, M. Benaissa, and D. Taylor, "On-Line Error Detection for Bit-Serial Multipliers in GF(2m)," in *J. Electronic Testing*, vol. 13, no. 1, 1998, pp. 29–40.

[7] C. H. Kim and J.-J. Quisquater, "Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures," in *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems, First IFIP TC6 / WG 8.8 / WG 11.2 International Workshop*, WISTP 2007, LNCS 4462. Springer, 2007, pp. 215–228.

[8] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Advances in Cryptology*, CRYPTO '96, LNCS 1109. Springer, 1996, pp. 104–113.

[9] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology*, CRYPTO '99, LNCS 1666. Springer, 1999, pp. 388–397.

[10] T. Malkin, F.-X. Standaert, and M. Yung, "A Comparative Cost/Security Analysis of Fault Attack Countermeasures," in *Fault Diagnosis and Tolerance in Cryptography*, FDTC 2006, LNCS 4236. Springer, October 2006, pp. 159–172.

[11] M. Medwed and J.-M. Schmidt, "Coding Schemes for Arithmetic and Logic Operations - How Robust Are They?" in *10th International Workshop on Information Security Applications*, WISA 2009, LNCS 5932. Springer, 2009, pp. 51–65.

[12] T. Rao, "Biresidue Error-Correcting Codes for Computer Arithmetic," *Computers, IEEE Transactions on*, vol. C-19, pp. 398–402, May 1970.

[13] H. J. Reinheimer, "Error Detecting and Correcting System and Method," U.S. Patent 3,699,323, October 17, 1972.

[14] M. M. Schaffer, "Residue Checking Apparatus for Detecting Errors in Add, Subtract, Multiply, Divide and Square Root Operations," U.S. Patent 4,926,374, May 15, 1990.

[15] J.-M. Schmidt, M. Hutter, and T. Plos, "Optical Fault Attacks on AES: A Threat in Violet," in *Fault Diagnosis and Tolerance in Cryptography*, FDTC 2009, IEEE-CS Press, September 2009, pp. 13–22.

[16] J.-M. Schmidt and M. Medwed, "A Fault Attack on ECDSA," in *Fault Diagnosis and Tolerance in Cryptography*, FDTC 2009, IEEE-CS Press, September 2009, pp. 93–99.

[17] S. P. Skorobogatov, "Semi-invasive attacks - A new approach to hardware security analysis," Ph.D. dissertation, University of Cambridge - Computer Laboratory, 2005.

[18] S. P. Skorobogatov and R. J. Anderson, "Optical Fault Induction Attacks," in *Cryptographic Hardware and Embedded Systems*, CHES 2002, LNCS 2523. Springer, 2003, pp. 2–12.