Sequential Circuit Design for Embedded Cryptographic Applications Resilient to Adversarial Faults

Gunnar Gaubatz, *Member*, *IEEE*, Erkay Savaş, *Member*, *IEEE*, and Berk Sunar, *Member*, *IEEE*

Abstract—In the relatively young field of fault-tolerant cryptography, the main research effort has focused exclusively on the protection of the data path of cryptographic circuits. To date, however, we have not found any work that aims at protecting the control logic of these circuits against fault attacks, which thus remains the proverbial Achilles' heel. Motivated by a hypothetical yet realistic fault analysis attack that, in principle, could be mounted against any modular exponentiation engine, even one with appropriate data path protection, we set out to close this remaining gap. In this paper, we present guidelines for the design of multifault-resilient sequential control logic based on standard Error-Detecting Codes (EDCs) with large minimum distance. We introduce a metric that measures the effectiveness of the error detection technique in terms of the effort the attacker has to make in relation to the area overhead spent in implementing the EDC. Our comparison shows that the proposed EDC-based technique provides superior performance when compared against regular N-modular redundancy techniques. Furthermore, our technique scales well and does not affect the critical path delay.

Index Terms—Data encryption, sequential circuits, fault tolerance, error control codes, error checking, adversarial faults.

1 Introduction

The ubiquitous nature of emerging cryptographic devices brings with it new threats and attack scenarios. The leakage of secret information through implementation specific side channels can be used by attackers to defeat otherwise impenetrable systems. Over the last decade, a large body of industrial, as well as academic, research has been devoted to the study of side channel attacks. In terms of high-level classification, one has to distinguish between active and passive attacks. Most of the R&D effort to date has been focused on the development of countermeasures against passive attacks, although, early on, Boneh et al. [1] and also Joye et al. [2] effectively demonstrated the strong need for the protection of various public-key systems against active attacks. Biham and Shamir [3] and, later, Piret and Quisquater [4] demonstrated successful fault attacks against the data paths of various block ciphers, whereas only recently have attacks on the control logic of block ciphers been reported by Choukri and Tunstall [5]. Current research on fault attack countermeasures is targeted primarily at techniques to reduce side-channel leakage, for example, in [6], and, secondarily, at increasing the fault tolerance of cryptographic devices—two goals that complement each other.

Most of the latter work is based on applying concurrent error detection (CED) techniques to the data path [7], [8]. Other techniques make use of architectural features such as the presence of both encryption and decryption units to compute the inverse operation for comparison of the result. Although this strategy works well with symmetric key algorithms such as the Advanced Encryption Standard (AES) [9], it may not work in a public-key setting, for example, if the private key is not available to check the result of an encryption [10].

We are currently unaware of any strategies to also protect the control logic (state machine, sequencer, execution pipeline, and so forth) of cryptographic systems against interference from the outside. A literature search revealed that there are several decades worth of research on the design of fault-tolerant state machines in classic application domains like automation control, avionics, and space-borne systems [11], [12], [13]. Although it is true that many of those findings could also apply to cryptographic systems, our main concern is that the fault model is fundamentally different. Fault-tolerant digital systems are typically designed around the premise that faults only occur one at a time and that there is sufficient time to recover between faults. Thus, all that is needed to build a fault-secure system is the ability to recover from the set of single faults [14]. Such an assumption seems reasonable as long as the faults are caused by stochastic events like "mother nature's" background radiation or the failure of components over time. In a cryptographic setting, we deal with faults of an adversarial nature, caused by an intelligent attacker who can be assumed to know about the structure and, thus, certain weaknesses of the system.

In this paper, we want to protect a system in a worst-case scenario, rather than maximizing average-case reliability. As such, it is paramount to not leave the proverbial

- *G. Gaubatz and B. Sunar are with the Electrical and Computer Engineering Department, Worcester Polytechnic Institute, ECE Atwater Kent Labs, Worcester, MA 01609. E-mail: {gaubatz, sunar}@wpi.edu.*
- *E. Savaş is with the Faculty of Engineering, Sabanci University, Orhanli, Tuzla, Istanbul, Turkey, TR-34956. E-mail: erkays@sabanciuniv.edu.*

Manuscript received 7 Dec. 2005; revised 15 Aug. 2006; accepted 8 Jan. 2007; published online 23 July 2007.
Recommended for acceptance by Ç. Koç.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0435-1205.
Digital Object Identifier no. 10.1109/TC.2007.70784.

Achilles' heel uncovered but to provide each and every component of the system with a uniform level of resilience against attacks. A secondary goal is to quantify this resilience and provide some practical design considerations. We would like to briefly remark on the issue of complexity and cost of implementation: Compared to the data path, the control logic tends to be fairly small in size. The overhead is therefore secondary to the improvement in fault resilience and, in most cases, even irrelevant. That said, we will show in Section 5 that our solution exhibits much better performance when compared to traditional fault-tolerant techniques of the same resilience level. Specifically, the contributions in this paper may be summarized as follows:

- We define a fault model that applies in an adversarial setting and provide a lengthy discussion on how realistic attacks targeting the control unit can be mounted.
- We define a useful metric for measuring the effectiveness of the error detection technique in terms of the effort the attacker has to make to mount a successful attack with respect to the area overhead spent in building the Error-Detecting Code (EDC).
- We provide a detailed analysis of the complexity of the proposed EDC and show that it provides superior error detection capabilities when compared against traditional redundancy-based techniques, while not affecting the critical path delay.
- We provide evidence that our technique provides better scalability compared to traditional redundancy-based techniques.

The remainder of this paper is structured as follows: Section 2 contains an example of a fault attack on the control circuit of a hypothetical modular exponentiation-based public-key accelerator. Section 3 will introduce the notation and definitions of sequential circuits, as well as classes of faults. State encoding schemes based on EDCs are introduced in Section 4 and used in Section 5 to design and analyze fault-resilient sequential circuits, complete with an example. Section 6 quantifies the error detection capabilities of various linear codes and explains the importance of design diversity to counter certain classes of faults.

2 Motivation

We will motivate our research with an example attack scenario. We will demonstrate the theoretical feasibility of an attack on a slightly simplified public-key cryptographic accelerator as it might be found in a ubiquitous security device such as a smart card. For the sake of simplicity, let us assume a basic modular exponentiation algorithm without CRT, using the secret private key, for example, an RSA decryption or signature generation. It should, however, be straightforward to adapt the attack to a CRT-based implementation. We consider a regular standard-cell ASIC implementation with sufficient protection of the data path but not the control logic, along with countermeasures to prevent timing and power analysis (balanced power consumption, constant runtime). We furthermore assume that the attacker has the ability to unpack the chip and induce bit flips on the state registers with some temporal precision [15], even though it is our understanding that tamper-proof coating and/or packaging has become something of a standard practice among smart card manufacturers. The work of Anderson and Kuhn [16] gives reason to question the effectiveness of such measures against determined attackers.

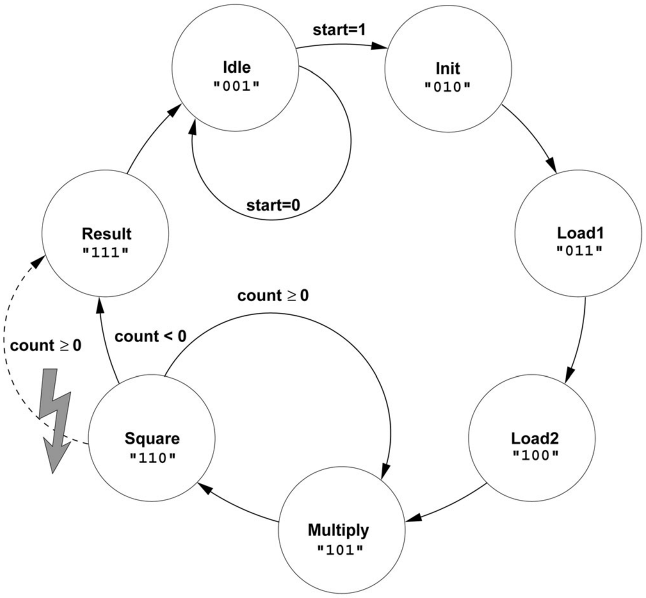

Fig. 1. State diagram representation of the Montgomery ladder algorithm with point of attack.

A particular algorithm that meets the requirements set above is Joye and Yen's [6] variant of the Montgomery ladder (Algorithm 1), computing $y = x^e \bmod N$. The state machine in Fig. 1 implements the algorithm on a data path with a single multiplier and three registers, R_0, R_1, and R_2. Each register is n bits wide, the size of the modulus N. This highly simplified state machine consists of seven states: the six active states, which are listed in Algorithm 1 right beside their respective instruction, and the idle state in which the circuit is inactive. The initial state after activation of the circuit sets up constants and counters. The two load states read the variables x and e via the input bus, whereas the result state returns the computed value y on the output bus. The multiply and square states are self-explanatory from the algorithm description. The control signals a and b for selecting between registers R_0 and R_1 are determined by scanning the exponent in R_2 bitwise, as indicated by the superscript bit index in parentheses.

Algorithm 1. Joye and Yen's Montgomery Ladder Exponentiation [6]

```
Input: $x$, $e$
Output: $y = x^e \bmod N$
  $\mathrm{R}_0 \leftarrow 1$                                   INIT
  $\mathrm{R}_1 \leftarrow x$                                   LOAD1
  $\mathrm{R}_2 \leftarrow e$                                   LOAD2
  for $i = n - 1$ downto 0 do
    $a \leftarrow \mathrm{R}_2^{(i)};\ b \leftarrow \neg a$
    $\mathrm{R}_b \leftarrow \mathrm{R}_b \cdot \mathrm{R}_a \bmod N$              MULTIPLY
    $\mathrm{R}_a \leftarrow \mathrm{R}_a^2 \bmod N$                   SQUARE
  end for
  $y \leftarrow \mathrm{R}_0$                                   RESULT
```

The complete public-key accelerator circuit consists of a modular multiplication unit, a memory block for the storage of operands and temporary results, and the exponentiation state machine, which coordinates the movement of data between the multiplier and memory. The multiplier can also be used for squaring operations. Although we will mainly focus on the state machine itself, we want to give a brief overview of the multiplier component for a more complete picture. It is a pipelined scalable Montgomery multiplier, as described in [17], which can perform arithmetic on both integer and binary polynomial operands. Scalable here refers to the ability to perform arithmetic with operands of nearly arbitrary precision only limited by the amount of memory, as well as the design time configurability of parameters like the number of pipeline stages and data path width. The multiplier computes the Montgomery product $C = A \cdot B \cdot R^{-1}$ using the word-level "finely integrated operand scanning" (FIOS) algorithm, which consists of two nested loops. To reflect a typical application scenario in an embedded system, we chose to implement a rather small variant of the multiplier with an 8-bit data path and three pipeline stages. Each stage consists of two 8×8 bit parallel multipliers and two adders and can process one complete inner loop of the algorithm. Intermediate results are computed LSW first and can be passed from stage to stage. Delay registers with bypass logic maintain proper scheduling of data words between the pipeline stages. Each pipeline stage has its own inner loop state machine and the scheduling of each stage is controlled by a single outer loop state machine. Since, typically, there are fewer pipeline stages than outer loop iterations, a FIFO stores the result of the last stage and passes it back to the first stage when it becomes available again. Table 5 in Section 5 gives the exact area breakdown of the multiplier components.

We would like to point out that the multiplier currently has no fault detection or other side-channel attack countermeasures. We use it mainly to demonstrate the rather small percentage of control logic in relation to the overall area of the design. Therefore, the total increase in area will stay relatively small, even if the error detection mechanism in the control logic adds significant overhead. This will become visible later on in the synthesis results of our example.

Our main argument, however, is the following: Even if the data path of the circuit were protected by attack countermeasures, the system would still be vulnerable to fault attacks unless the state machine is protected by an error detection mechanism. The following is a brief outline of such a hypothetical attack, which reveals the n-bit secret exponent e in a short time:

1. Measure the total time of an (arbitrary) exponentiation and determine the computation time required per exponent bit: $t_{\mathrm{bit}} \approx t_{\mathrm{total}}/n$.
2. Set the round index $k = 1$, select a known ciphertext x and set the result of the previous round as $(\mathrm{R}_0, \mathrm{R}_1) = (1, x)$.
3. Compute the triple $(A, B, C) = (\mathrm{R}_0^2, \mathrm{R}_0 \cdot \mathrm{R}_1, \mathrm{R}_1^2) \bmod N$ and make a hypothesis H_0 about the exponent bit $e^{(n-k)} \in \{0, 1\}$.
4. Start exponentiating x.
5. After $k \times t_{\mathrm{bit}}$ time steps, force the state machine into the state `111:RESULT` by inducing a fault either on the least significant bit of the state or on the input signal `count` < `0`. This will cause the circuit to reveal its intermediate result in R_0.
6. Verify the hypothesis by comparing the actual value of R_0 to A. If it matches $(e^{(n-k)} = 0)$, set $(\mathrm{R}_1, \mathrm{R}_0) = (A, B)$; else, if it matches $(e^{(n-k)} = 1)$, set $(\mathrm{R}_0, \mathrm{R}_1) = (B, C)$.
7. Repeat Steps 3 through 6 for all other bits of the exponent $(k = 2 \dots n)$.

This attack works because there is no way to distinguish between a valid or a faulty transition (cf., Fig. 1: solid versus dashed arc) from the `110:SQUARE` state to the final state `111:RESULT`. Without a doubt, an actual attack on real hardware will probably be much more involved and not as simple as described here in our naive scheme. Our basic point is, however, that the control logic of any cryptographic circuit should be identified as a potential point of attack. This is especially true if other countermeasures against fault attacks on the data path are put into place since the control logic becomes the weakest element in the system and, as such, an obvious target. To this end, we propose an all-encompassing strategy to protect all of the components of embedded cryptographic devices with a certain minimum level of guaranteed resilience to fault attacks.

3 Preliminaries and Definitions

A finite-state machine (FSM) can be formally defined by a six-tuple $(S, I, O, \Delta, \Lambda, \mathbf{s}_0)$. S is the set of valid states (the state space), which has dimension $k = \lceil \log_2 |S| \rceil$, and $\mathbf{s} \in S$ denotes the current state. In a similar manner, I and O define the input and output sets and $\mathbf{i} \in I$ and $\mathbf{o} \in O$, the current inputs and outputs. The number of input and output signals are $\iota = \lceil \log_2 |I| \rceil$ and $\omega = \lceil \log_2 |O| \rceil$ bits. The sets Δ and Λ represent the next-state and output logic functions, which take $\mathbf{s}$ and $\mathbf{i}$ as their arguments. They consist of a collection of Boolean functions $\delta_j(\mathbf{s}, \mathbf{i})$ and $\lambda_l(\mathbf{s}, \mathbf{i})$, each of which compute a single bit in the next state or output vector. Finally, $\mathbf{s}_0$ indicates the initial state. For simplicity and without loss of generality, we will assume that all 2^k states are valid throughout the remainder of this paper. In the following, we will refer to state machines that encode the state in a vector with $n > k$ bits as redundant state machines.

A regular (read: non-fault-resilient) state machine (Fig. 2) encodes the current state as a k-bit binary vector $\mathbf{s} = (s_0, s_1, \dots, s_{k-1})$. In linear algebra terminology, the state space S is a vector space over the field GF(2), spanned by a set of k linearly independent basis vectors $\nu_0 \dots \nu_{k-1}$. The current state is a linear combination $\mathbf{s} = s_0\nu_0 + s_1\nu_1 + \dots + s_{k-1}\nu_{k-1}$, with the addition in GF(2) being equivalent to an exclusive OR operation. In the simplest case, the basis vectors form a standard basis with orthogonal unit vectors:

$$\begin{array}{rcl}
\nu_0 & = & (1 \;\; 0 \;\; \dots \;\; 0) \\
\nu_1 & = & (0 \;\; 1 \;\; \dots \;\; 0) \\
\vdots & & \;\vdots \;\; \vdots \;\; \ddots \;\; \vdots \\
\nu_{k-1} & = & (0 \;\; 0 \;\; \dots \;\; 1).
\end{array}$$

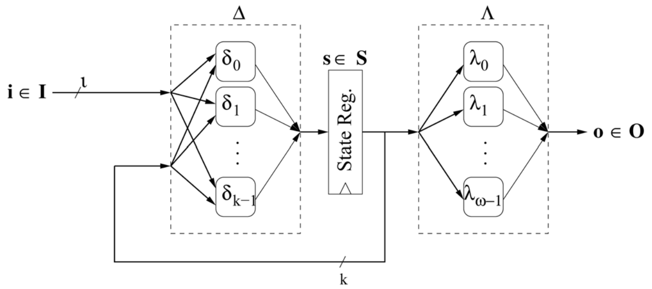

Fig. 2. Nonredundant Moore-style state machine with ι inputs, 2^k states, and ω outputs.

The next state $\mathbf{s}'$ is determined by evaluating the set of next state functions:

$$\mathbf{s}' = \boldsymbol{\Delta}_k(\mathbf{s}, \mathbf{i}) = (\delta_0(\mathbf{s}, \mathbf{i}), \delta_1(\mathbf{s}, \mathbf{i}), \ldots, \delta_{k-1}(\mathbf{s}, \mathbf{i})).$$

Similarly, the ω-bit output vector $\mathbf{o}$ is computed by evaluating the set of output logic functions $\Lambda_\omega(\mathbf{s}) = (\lambda_0(\mathbf{s}), \lambda_1(\mathbf{s}), \ldots, \lambda_{\omega-1}(\mathbf{s}))$. At each clock edge, the current state vector takes on the value of the next state vector $\mathbf{s} = \mathbf{s}'$. Mealy type state machines are different only in that their output functions also depend on the input vector. In the following, we will focus on Moore-type machines, but the same principles also apply to Mealy machines.

3.1 Classes of Faults

We would like to distinguish between different classes of faults. A *common-mode failure* (CMF) [18] is a set of multiple simultaneously occurring faults resulting from a single cause, for example, due to a glitch in the power supply or a large area ionization of an unpackaged integrated circuit with a radiation source. Highly regular circuits made from replicated components are the most prone to CMF if the cause of the fault is not one of limited locality. In contrast, a *single-mode failure* (SMF) is a single fault that occurs due to a single cause. Oftentimes, the effect of the fault is locally limited. Examples of this type of fault are single stuck-at faults caused early on by manufacturing flaws or later through a breakdown resulting from electromigration. In an adversarial setting, a transient SMF may also be induced by means of a "light attack". This refers to the introduction of single bit errors into an unpackaged integrated circuit by means of a highly focused laser beam or other ionization source [15]. It requires a high degree of spatial and, potentially, also temporal precision.

3.2 Adversarial Fault Model

An active side channel attack such as differential fault analysis (DFA) relies on the manifestation of injected faults as erroneous results, which can then be observed at the output of the device. The error is therefore the difference between the expected output x and the observed output $\tilde{x} = x + e$. In the following, we do not assume that the adversary is limited to any specific method of fault injection. The only assumption is that direct invasive access to the chip itself is prevented by some tamperproof coating, a reasonable assumption since this is a common practice, for example, in the smart card industry.

When talking about errors as the manifestations of faults, there are two principal ways of characterization. A logical error is a bitwise distortion of the data, usually modeled as the XOR of data and error, that is, $\tilde{x} = x \oplus e$, whereas arithmetical errors admit the propagation of carries up to the range limit: $\tilde{x} = x + e \bmod 2^k$, where k is the width of the data path. The former is appropriate for storage-dominated devices (register files, RAM, flip-flops, and so forth); the arithmetic error model is more useful for arithmetic circuits such as adders and multipliers. Naturally, we will use the logical error model throughout the remainder of this paper since it is the most appropriate.

In the following, we will assume that an attacker is somehow able to induce a fault into the device that results in the flipping of a bit. The attacker has control over the location of the bit flip, but there is a cost (or effort) attached which increases with the number of bit flips. For example, if the attacker tries to change the state of a circuit by attacking the state register such that $\tilde{\mathbf{s}} = \mathbf{s} \oplus e$, the cost of the attack depends on the hamming weight of the induced error pattern e.

3.3 A Novel Effectiveness Metric: Attack Effort per Area

We need a metric by which to compare the effectiveness of various error detection mechanisms. An intuitive approach is to compare the cost or effort of a successful fault attack to the cost of preventing it. Before we can define this metric, we need to introduce a couple of further definitions:

- *Resilience* is a measure of how many errors the circuit can withstand. More concretely, this is the number of errors that can be accurately detected by the error detection network. Throughout this paper, we will refer to resilience as the parameter t and to a fault-tolerant sequential circuit as t-resilient.
- *Effort* is a measure for the difficulty in inducing a fault into the system that manifests itself as one or more errors. It cannot be given in absolute terms since an attacker's effort depends on a large variety of factors. We will therefore only give the effort in relative terms, compared to the cost for induction of a single bit error. There exist multiple types of faults, all of which warrant their own effort values. For simplicity, we only model single bit faults, which also manifest themselves as single bit errors at the gate level.

We now define the total attack cost as the sum of the total effort required for a successful attack. In this paper, we arbitrarily assign a nominal effort of 100 percent for a single bit error as the basis of comparison for various countermeasures. If the circuit is more resilient, then the effort for a successful attack also increases.

Definition 1. *Let E_{bf} denote the effort required for inducing a change of value for a single bit in the circuit (bit fault). Let t denote the degree of the worst-case fault resilience of the circuit, that is, the maximum number of errors that can be tolerated when induced simultaneously. The total cost for the attack is therefore $C = (t+1) \cdot E_{\mathrm{bf}}$.*

In Definition 1, we implicitly assume that the fault insertion effort increases at least linearly with resilience t. Inducing change of values for more than 1 bit in the circuit may not be much harder than changing 1 bit, especially when those bits are stored in places close to each other (for example, several consecutive bits in a register). However, our proposed scheme necessitates changing a number of certain bits, which are not particularly close to each other, if the fault induction attack is to be successful. Many successful fault induction attacks do not necessarily need a high spatial precision and inadvertently changing several other bits than the target bits is acceptable in these attack scenarios. The proposed technique, on the other hand, forces the attacker to be very precise when changing certain bits lest the attack be detected. Therefore, changing several bits, not necessarily next to each other in the circuit, without changing other bits is indeed a daunting task for the attacker. The exact relation of fault insertion effort to the resilience t largely depends on the specific implementation and, therefore, is difficult to formulate. Nevertheless, our assumption of a linearly increasing effort for fault insertion with respect to t may be justifiable, at least as a first-order approximation, since it is our intuition that the relation is even more complex and the effort should increase faster than just linearly.

Based on Definition 1 and the area overhead that is required to implement a specific countermeasure, we can define an effectiveness metric η that allows us to compare the different approaches. The figure of merit is the ratio between the cost and area. Additionally, we define a normalized effectiveness H that can be useful for comparing countermeasures on circuits with completely different functionality.

Definition 2. *Let A denote the circuit area and C the cost for successfully implementing a fault attack. The effectiveness ratio $\eta = C/A$ weighs the cost of the attack against the cost of the countermeasure (circuit area). Let $\eta_0 = C_0/A_0$ be the effectiveness of the unprotected circuit. Then, $H = \eta/\eta_0$ denotes the normalized effectiveness of a countermeasure with respect to a particular basis circuit.*

In Section 5, we will use the effectiveness metric to compare traditional fault-tolerant approaches with our method.

4 Redundant State Encoding Based on EDCs

Looking at typical FSM implementations of cryptographic algorithms, one may notice that, oftentimes, only relatively few different states are necessary or, if a more complex algorithm is required, the control logic can be broken down hierarchically. Typically, only a few bits are necessary to store the state, depending on the encoding scheme used. Compared to the size of the data path in a typical embedded cryptographic system, the size of the control logic is often relatively small, even when a highly redundant encoding scheme is being used. As an example, we refer to the Montgomery multiplier introduced above, which has a total area of 10,410 equivalent gates, excluding the area for data storage. The control logic occupies an absolute area of 2,015 equivalent gates, around 20 percent; the remaining 80 percent belongs to the data path.

Electronic design automation tools often allow specification of a preferred state encoding scheme for the automatic synthesis of FSMs. The typical styles to choose from are "one-hot," "gray," or "binary" encoding, but they only allow a trade-off between the speed and the area of the resulting circuit. We propose investigating a third trade-off alternative, that is, the degree of fault tolerance or, put differently, the resilience against fault attacks. Although fault-tolerant sequential circuits have been the subject of research in the context of reliable system design, there are a couple of important aspects that require further investigation in the context of cryptographic circuits due to the fundamentally different adversarial fault model.

We place a special emphasis on the detection, rather than correction, of faults. Our argument is quantitative in that the number of detectable faults $(d-1)$ is larger than the number of correctable faults $(\lfloor(d-1)/2\rfloor)$, based on the minimum Hamming distance d of the coding scheme. Since there is no way to tell whether the multiplicity of the fault is strictly less than $d/2$ or not, there is a chance that the error correction will produce a valid but ultimately incorrect next state. On the other hand, we also argue qualitatively: Due to the adversarial nature of faults in the cryptographic context and their potentially devastating effects on the security of the system, the detection of errors is far more important than producing a result under all circumstances. Detected faults should, rather, be dealt with under a suitable exception handling mechanism that prevents side-channel leakage, for example, by erasing a sensitive key material. As a side benefit, the area overhead for detection is typically much less than that of correction and the circuit does not have to sit on the critical path.

One-hot encoding is preferred by digital logic designers whenever speed and a simple next-state logic are desired. Despite its large amount of redundancy, its minimum-distance properties are weak since valid code words only differ by two bits. In the case of a double error that resets the active bit and sets another one, there is not enough information in the encoding to detect the error and, thus, one-hot encoding is not suitable for our purposes. Since performance is less of a priority in security-critical applications, we can trade off the speed of one-hot encoding for the capability to detect errors. One way of doing so is by choosing a state encoding based on EDCs with large minimum distance. For example, if we had a state machine with $m = 15$ states, one-hot encoding would require a 15-bit state vector. Encoding the states with a (15, 4)-Simplex code, which is the dual of a (15, 11)-Hamming code, would require a state vector of exactly the same size. One-hot encoding only has a minimum distance of 2, whereas the Simplex code has a distance of 8, which allows detection of all errors with a Hamming weight of up to 7.

5 Fault-Resilient Sequential Circuits

A fault-resilient state machine (Fig. 3) must necessarily incorporate redundancy—not only in the state encoding, but also in the combinational logic of next-state and output functions. In a state machine protected by a linear (n, k) code,

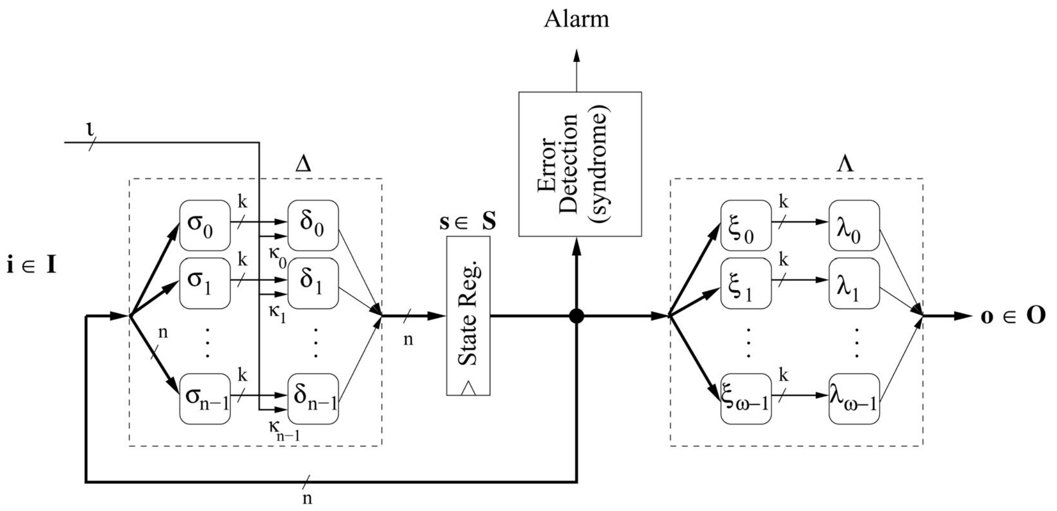

Fig. 3. Fault-resilient Moore-style state machine.

the redundant state vector $\mathbf{s}^{(n)} = (s_0, s_1, \ldots, s_{n-1})$ now consists of n bits. For the remainder of this paper, we will also implicitly assume that the input signal $\mathbf{i}$ is given in redundant form as well. Its next-state logic is a set $\boldsymbol{\Delta}_n(\mathbf{s}, \mathbf{i})$ of n functions, δ_j, that each determine one bit of s. In general, since there are only 2^k states (but encoded in redundant form), each δ_j is a Boolean function over $k + \iota_k$ variables (ι_k denotes the number of nonredundant bits in the input vector $\mathbf{i}$).

Formally speaking, the state space $S^{(n)}$ is a k-dimensional subspace of an n-dimensional vector space $\{0,1\}^n$. The basis vectors of this subspace are the n-bit row vectors of the generator matrix $\mathbf{G}$ of the code. Any value of the state vector s that is not a linear combination of these basis vectors is therefore treated as an error. Although this interpretation nicely illustrates the mapping of k-bit states into a redundant n-bit form, it is not useful for the definition of next-state functions because we want to keep the state in redundant form only. If we were to decode the current state, compute the next state in only k bits, and reencode it to n bits, this would leave the system with a single point of failure in the next-state function, rendering it vulnerable to attacks. Rather, we keep the system state in redundant form at all times to enhance its resilience against attackers probing for the weakest spot of the system.

An alternative interpretation that is more suitable for our definition of a redundant state machine is to view the columns of $\mathbf{G}$ as n redundant basis vectors of the nonredundant k-dimensional state space S, which is isomorphic to $S^{(n)}$. Any k *linearly independent* basis vectors form a complete basis for S and span the entire state space. Thus, those k (out of n) state variables that are associated with the respective vectors of such a basis can be used for specification of a next state or output function. In the following, we will let σ_j and ξ_j denote such sets of k state variables associated with linearly independent basis vectors (columns of $\mathbf{G}$) for the next state functions δ_j and output functions λ_j, respectively. Additionally, we introduce nonredundant subsets κ_j of input variables (one per next state function). Each κ_j selects ι_k input variables from the redundant input vector $\mathbf{i}$. If the columns of $\mathbf{G}$ were not linearly independent, then this would introduce an ambiguity about the current state and certain functions could not be implemented. It usually depends on the exact construction of the code to determine which of all $\binom{n}{k}$ possible sets of k state variables are suitable.

If enough many sets $\sigma_j, \kappa_j, \xi_j$ are available, then the next state and output logic functions can be defined over a variety of such sets. It is prudent to make use of such alternative definitions so as to minimize the hazardous effect that a single faulty state variable has on the overall system.

Example. An example will help to illustrate the last point: Let $\mathbf{G}$ be the generator matrix of a systematic (7, 3)-Simplex code (dual of the (7, 4)-Hamming):

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

The rows of $\mathbf{G}$ form the k-dimensional basis for the state space S as a subspace of $\{0,1\}^n$. Let $\mathbf{s}^{(n)} = \mathbf{s}^{(k)}\mathbf{G}$ be the redundant state vector. Using the modular exponentiation example in Section 2, the states of a redundant state machine would be encoded as in Table 1. Alternatively, we can interpret the columns of $\mathbf{G}$ as redundant basis vectors $\nu_0, \nu_1, \ldots, \nu_6$ spanning S and associated with $s_0, s_1, \ldots, s_6$, respectively.

TABLE 1
Simplex State Encoding
for the Modular Exponentiation Example

State	$\mathbf{s}^{(k)} = (s_0, s_1, s_2)$	$\mathbf{s}^{(n)} = (s_0, s_1, s_2, s_3, s_4, s_5, s_6)$
Idle	001	0010111
Init	010	0101011
Load1	011	0111100
Load2	100	1001101
Multiply	101	1011010
Square	110	1100110
Result	111	1110001

In a nonredundant state machine, the next state logic functions would be specified as $\delta_j(\mathbf{s}^{(k)}, \mathbf{i})$. In the redundant case, we still have the same number of states, but, now, we have $n = 7$ state variables $(s_0, s_1, \ldots, s_6)$ that we can use to specify the next state functions. Due to systematic encoding, we have the following relationships between redundant $(s_0, s_1, s_2, s_3, s_4, s_5, s_6)^{(n)}$ and nonredundant state variables $(s_0, s_1, s_2)^{(k)}$:

$$\begin{aligned} s_0^{(n)} &= s_0^{(k)}, \\ s_1^{(n)} &= s_1^{(k)}, \\ s_2^{(n)} &= s_2^{(k)}, \\ s_3^{(n)} &= s_0^{(k)} \oplus s_1^{(k)}, \\ s_4^{(n)} &= s_0^{(k)} \oplus s_2^{(k)}, \\ s_5^{(n)} &= s_1^{(k)} \oplus s_2^{(k)}, \\ s_6^{(n)} &= s_0^{(k)} \oplus s_1^{(k)} \oplus s_2^{(k)}. \end{aligned}$$

For example, let $\sigma_j = (s_1, s_3, s_6)$. Since columns 1, 3, and 6 of $\mathbf{G}$ are linearly independent, it is generally possible to specify an arbitrary next-state or output function over this set of state variables, for example, $\delta_j(\sigma, \mathbf{i}) = \delta_j(s_1, s_3, s_6, i_0)$. This is always possible and does not depend on what the actual application of the state machine is.[1] The Boolean function definition of δ_j, however, does depend on the application.

Conversely, it should immediately become clear that we should not specify a next-state function over the sets (s_0, s_1, s_3) or (s_0, s_2, s_4). The bases associated with the selection of state variables are $\{\nu_0, \nu_1, \nu_3\} = \{(1,0,0), (0,1,0), (1,1,0)\}$ and

$$\{\nu_0, \nu_2, \nu_4\} = \{(1,0,0), (0,0,1), (1,0,1)\},$$

both of which are not linearly independent: $\nu_0 + \nu_1 + \nu_3 = \mathbf{0}$ and $\nu_0 + \nu_2 + \nu_4 = \mathbf{0}$ in GF(2). In both examples, one coordinate is always zero in all basis vectors. If a hypothetical next state function depends on that coordinate to distinguish between two different states, the function cannot be implemented.

We will now give a concrete example of a fault-resilient state machine implementation using the error detecting (7, 3)-Simplex code. We will further compare our approach to N-modular redundancy, a concept from traditional fault-tolerant computation, with respect to area overhead and fault resilience. In N-modular redundancy, functional modules are replicated N times, ideally employing design diversity techniques. Their outputs are either voted on by a majority logic (that is, for error correction) or simply compared for consistency to detect any error that may occur in a module. The most common configuration is *triple modular redundancy* (TMR), sometimes also referred to as triads [19], which allows either the detection of two errors or the correction of a single error. Since a fault-resilient system requires redundancy not only in the state machine but also overall, redundant versions of the input and output signals need to be present as well. In a system with nonredundant inputs, for example, the attack described in Section 2 could be carried out by forcing the condition `count` < 0, which may be indicated to the state machine by a single status bit. Such an attack is harder to accomplish if input signals are available as multiple redundant and independent variables. The amount of redundancy should be chosen to protect against the same number of errors t as the state encoding scheme to ensure a uniform level of resilience for the entire system. The level of resilience is measured by the minimum number t of signals an attacker needs to change (bit flips) for a successful attack. For state encoding with linear codes over GF(2), we have $t = d - 1$, where d is the minimum distance (or nonzero Hamming weight) of all code words. In a modular redundant scheme, the overall resilience level can be found by concatenating the state vectors of the individual modules and computing its total minimum distance, D, and resilience, $t = D - 1$. They are determined by the degree of modular redundancy, N, and the distances, d_j, of each module's individual state encoding:

$$D = \sum_{j=1}^{N} d_j.$$

In the following, we will therefore require a D-fold redundant set of input signals to be available to the circuit, which in turn must provide a D-fold redundant set of output functions. This provides coverage for the corner case where all t faults occur on the inputs. For example, in the case of a TMR system with simple binary encoded states, the minimum distance between words of the concatenated state vectors is 3 $(D = N)$ with a worst-case resilience of $t = 2$. Our EDC-based state machine implementation is a special case, with $N = 1$, $d = 4$, and therefore requires an input set of distance 4 to maintain the same amount of resilience against $D - 1 = 3$ faults.

We will now provide an example using the Montgomery ladder algorithm introduced earlier in Section 2. A very simple state machine implementation requires two input signals, `start` and `count!` $= 0$, and the D-fold redundant input vector thus consists of the signals $\mathbf{i} = (i_{0,0}, \ldots, i_{0,D-1}, i_{1,1}, \ldots, i_{1,D-1})$, with the mapping $i_{0,j} =$ `start` and $i_{1,j} =$ `count!` $= 0$. It computes three output signals for controlling the data path (multiplier): `mul_op_a`, `mul_op_b`, and `mul_res`. Again, these signals will be generated with D-fold redundancy and mapped to the output vector $\mathbf{o} = (o_{0,0}, o_{0,1}, o_{0,2}, \ldots, o_{2,D-1})$.

As with the redundant state vector, we will not use all elements of $\mathbf{i}$ for definition of the next state functions, but only subsets κ_j, each containing two input variables. Finally, we need ω sets ξ_j of k state variables each as inputs to the output functions λ_j, again with minimal overlap. For our example, we chose:

[1] Once a specific function is defined, it might not need the complete state information in k variables and the logic implementation may be minimized.

TABLE 2
State Transition Table of the Fault-Resilient Modular Exponentiation FSM

State	s_0	s_1	s_2	s_3	s_4	s_5	s_6	start	count< 0	s'_0	s'_1	s'_2	s'_3	s'_4	s'_5	s'_6
(illegal)	0	0	0	0	0	0	0	x	x	0	0	0	0	0	0	0
idle	0	0	1	0	1	1	1	0	x	0	0	1	0	1	1	1
idle	0	0	1	0	1	1	1	1	x	0	1	0	1	0	1	1
init	0	1	0	1	0	1	1	x	x	0	1	1	1	1	0	0
load1	0	1	1	1	1	0	0	x	x	1	0	0	1	1	0	1
load2	1	0	0	1	1	0	1	x	x	1	0	1	1	0	1	0
mult	1	0	1	1	0	1	0	x	x	1	1	0	0	1	1	0
sqr	1	1	0	0	1	1	0	x	0	1	0	1	1	0	1	0
sqr	1	1	0	0	1	1	0	x	1	1	1	1	0	0	0	1
res	1	1	1	0	0	0	1	x	x	0	0	1	0	1	1	1

$$
\begin{array}{lllllll}
\sigma_0 & = & (s_0, s_2, s_3) & \kappa_0 & = & (i_{0,0}, i_{1,0}) \\
\sigma_1 & = & (s_1, s_3, s_4) & \kappa_1 & = & (i_{0,1}, i_{1,1}) \\
\sigma_2 & = & (s_2, s_4, s_5) & \kappa_2 & = & (i_{0,3}, i_{1,2}) \\
\sigma_3 & = & (s_3, s_5, s_6) & \kappa_3 & = & (i_{0,0}, i_{1,3}) \\
\sigma_4 & = & (s_0, s_4, s_6) & \kappa_4 & = & (i_{0,1}, i_{1,0}) \\
\sigma_5 & = & (s_0, s_1, s_5) & \kappa_5 & = & (i_{0,2}, i_{1,2}) \\
\sigma_6 & = & (s_1, s_2, s_6) & \kappa_6 & = & (i_{0,3}, i_{1,3})
\end{array}
$$

$$
\begin{array}{llllll}
\xi_0 & = & (s_0, s_1, s_2) & \xi_6 & = & (s_0, s_4, s_5) \\
\xi_1 & = & (s_0, s_3, s_5) & \xi_7 & = & (s_2, s_5, s_6) \\
\xi_2 & = & (s_3, s_4, s_6) & \xi_8 & = & (s_1, s_2, s_6) \\
\xi_3 & = & (s_2, s_3, s_4) & \xi_9 & = & (s_0, s_3, s_4) \\
\xi_4 & = & (s_1, s_2, s_4) & \xi_{10} & = & (s_1, s_5, s_6) \\
\xi_5 & = & (s_3, s_5, s_6) & \xi_{11} & = & (s_0, s_1, s_6).
\end{array}
$$

From the state transition table (Table 2), we can now derive the concrete next state functions based on the state and input variable sets σ_j and κ_j:

$$
\begin{array}{lll}
\delta_0(\sigma_0, \kappa_0) & = & s_0 s'_2 + s_2 s_3, \\
\delta_1(\sigma_1, \kappa_1) & = & i_{0,1} s'_3 s_4 (s_1 + i_{1,1}) + s'_1 s'_3 s_4 i_{1,1} + s_3 s'_4, \\
\delta_2(\sigma_2, \kappa_2) & = & s_2 s'_4 s'_5 + s'_2 s_4 + s'_2 s_5 + s_4 s_5 i'_{1,2}, \\
\delta_3(\sigma_3, \kappa_3) & = & s'_3 s_5 s'_6 i'_{0,0} + s_3 s'_5 + s_3 s_6 + s_5 s_6 i_{1,3}, \\
\delta_4(\sigma_4, \kappa_4) & = & s_0 s'_4 + s'_0 s_4 s'_6 + s'_0 s_4 i'_{1,0} + s'_4 s_6, \\
\delta_5(\sigma_5, \kappa_5) & = & i'_{0,2} s_0 + s_0 s'_5 + s'_1 s_5, \\
\delta_6(\sigma_6, \kappa_6) & = & i_{0,3} s_1 s'_6 + s_1 s_2 + s_2 s_6.
\end{array}
$$

We can derive the output logic equations in a similar fashion (omitted here due to space considerations).

5.1 Synthesis Results and Overhead Analysis

Due to the principal difference in error detection capability between an N-modular redundant and an EDC-based state machine, we decided to compare the nonredundant implementation to four different redundant implementations with slightly varying parameters:

1. a two-fault resilient triple modular redundant (TMR, $N = 3$) version,
2. a three-fault resilient quadruple modular redundant (QMR, $N = 4$) version,
3. an internally three-fault resilient EDC version with a threefold redundant I/O set (3-EDC),[2] and
4. a three-fault resilient EDC version with a fourfold redundant I/O set (4-EDC).

Our initial analysis consisted of using the University of California, Berkeley, SIS logic synthesis tool and the MCNC synthesis script for mapping the five different circuit descriptions to the MCNC standard libraries `mcnc.genlib` and `mcnc_latch.genlib` and comparing the results (Table 3).

5.1.1 Area Overhead

Despite the higher literal count of 3-EDC and 4-EDC compared to the TMR implementation, the actual circuit area is almost exactly the same. The lower resilience of TMR against faults, however, makes a compelling argument for building fault-resilient state machines based on EDCs, especially since the degree of resilience will be even better for a larger state space dimension k. A QMR design requires more than 100 percent more overhead for achieving the same level of resilience. In addition, while N-modular redundancy schemes have a constant overhead of approximately $(N-1) \cdot 100\%$ independent of the state space dimension k, the (storage) overhead of EDC-based fault-resilient state machines actually decreases with a larger state space (cf., Fig. 4). It appears reasonable to expect a similar trend for the complexity of the next-state logic.

These initial results encouraged us to proceed with the analysis of a more detailed state machine model synthesized using Synopsys DesignCompiler and a 0.13 μm ASIC standard cell library. Special care was taken to direct the synthesis not to share logic between the next state functions δ_n. This is to avoid the manifestation of a single stuck-at fault as an error on multiple outputs of the next state function $\Delta_n(\mathbf{s}, \mathbf{i})$. The results[3] are given in Table 4. It shows some clear differences in area requirements between the codes-based and the TMR implementation which are not visible from the initial analysis. To some extent, this is due to the more accurate modeling (we only used a simplified state machine design for the SIS flow), but the

[2] This effectively reduces the worst-case fault resilience to two faults on the input set.

[3] The area is given in terms of equivalent gate size, that is, the size relative to a two-input NAND gate.

TABLE 3
Initial Analysis of Various EDC and NMR Schemes

	Non-red	3-EDC	4-EDC	TMR	QMR
# Inputs	2	6	8	6	8
# Outputs	3	10	13	10	13
# SOP-Literals	42	155	155	135	179
Gate Count	19	56	60	59	77
# Latches	3	8	8	10	13
Total Area	64	207	210	210	276
Overhead (%)	0	223	228	228	331

main reason is the fact that we used an industrial strength standard cell library with advanced complex gates of varying drive strength. The size of these gates is often more compact than a realization from simple gates and is given in precise decimal fraction area numbers. The size of gates in the MCNC library, on the other hand, was only given in integers.

5.1.2 Effectiveness and Scalability

Table 4 also gives the performance of our examples in terms of the effectiveness metric for fault resilience measures that we defined in Definition 2 in Section 3. The value η describes the relative effort per gate, whereas H is the normalized effectiveness value relative to the effectiveness of the unprotected circuit. The results are quite interesting: As expected, the EDC-based fault-resilient circuits show the best effectiveness in preventing fault attacks; however, the NMR approach fares much worse than having no countermeasure at all. We also experimented with a slightly different implementation with 10 states and a much larger resilience, $t = 7$. Here, we used a (15, 4)-Simplex code for state encoding. Our results indicate that, at slightly more than 400 percent area overhead, the EDC-based scheme scales much better than NMR, for which we expect an overhead of well beyond 800 percent at the same resilience level.

5.1.3 Critical Path Overhead

We could not determine any substantial critical path overhead with any of our examples. All designs synthesized at 1 GHz clock frequency without problems and we would expect the critical path to be on the data path much rather than the control logic. In the case of 3-EDC and 4-EDC, the redundant state variables are computed in parallel, completely independent of each other, and therefore do not add to the critical path. Furthermore, the error detection network is not in the critical path either since it operates in parallel with the next-state and output logic. We conclude that there is no performance overhead associated with these fault resilience measures.

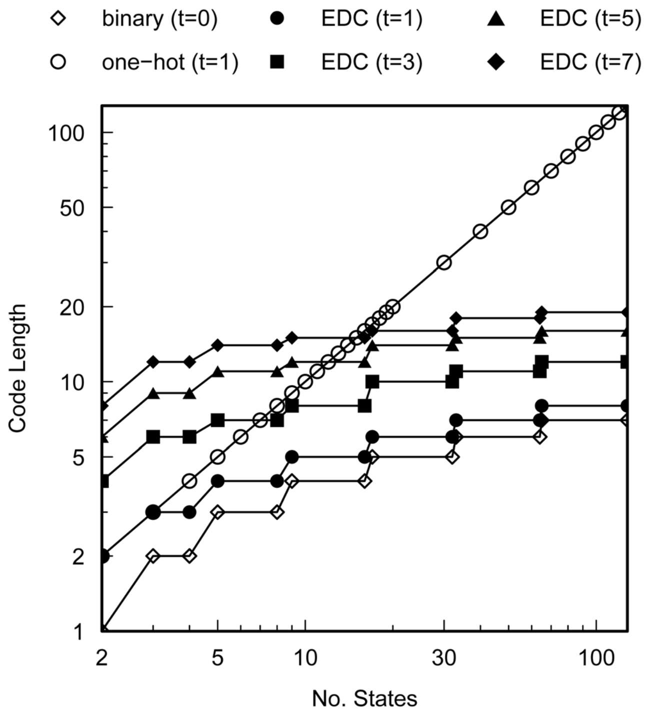

Fig. 4. Lengths of various t-resilient EDCs.

5.2 Influence of Control Logic Overhead on Total Circuit Area

Following the overhead analysis of the exponentiation state machine, we want to extrapolate the total overhead of implementing fault-resilient control logic throughout the entire modular exponentiation circuit. We start from the breakdown of area requirements for the Montgomery multiplier described in Section 2. Table 5 shows the components of the multiplier sorted by type and their size in equivalent gates. For extrapolation, we factored the overhead estimates in Table 4 into the area requirement for control logic but not for the data path. Due to the fact that control logic occupies less than 20 percent of the total area in a nonredundant implementation, the overhead due to fault resilience techniques is limited to around 40 percent for our method, whereas, for quadruple modular redundancy, it exceeds 80 percent (Table 6).

TABLE 4
Analysis of Detailed FSM Models (*10-State FSM with $k = 4$)

	Non-red	3-EDC	4-EDC	TMR	QMR	non-red.*	EDC
Total Area (eqv. gates)	48	137	143	189	249	71	361
Area Overhead	0.0%	188.1%	199.3%	295.8%	422.4%	0.0%	413.3%
Resilience (bit errors)	0	2	3	2	3	0	7
Attack effort	100%	300%	400%	300%	400%	100%	800%
Effectiveness η	2.098	2.184	2.804	1.590	1.606	1.422	2.216
norm. Effectiveness H	1.000	1.041	1.336	0.758	0.766	1.000	1.559

TABLE 5
Data Path/Control Logic Area Breakdown of the Montgomery Multiplier

Component (Type)	Area (gates)	Area (rel.)
Pipeline Stages (Data Path)	6,756	64.90%
Delay Registers (Data Path)	1,640	15.75%
Inner Loop FSM (Control Logic)	1,079	10.36%
Outer Loop FSM (Control Logic)	848	8.15%
FIFO control (Control Logic)	87	0.84%

6 Selection of Codes for t-Fault-Resilient State Machines

We have seen in the previous example how we can design a fault-resilient state machine with up to eight states using an ordinary (7, 3)-Simplex code. Since this code has a minimum distance of four, it is able to detect up to three faults in the state vector s. In general, the number of errors that can be detected by means of a linear block code with minimum distance d is $t = d - 1$. The selection of a particular code for a fault-resilient state machine implementation therefore depends on the desired level of resilience, that is, the number of detectable errors t, as well as the dimension of the state space k. These two parameters determine the length n of the block code, as shown in Table 7. We obtained our numbers from the reference table of minimum-distance bounds for binary linear codes [20]. We restructured the table in order to display the minimum length n for a given dimension k and desired fault resilience t.

The first column of the table consists entirely of trivial repetition codes because of $k = 1$. The first row, on the other hand, consists of all parity codes with no repetitions of column vectors, but with only a minimum distance of 2. These two are the extremes between which we can trade off the fault resilience versus the code length (or, rather, compactness). This is visualized in Fig. 4, where EDCs with different levels of fault resilience are juxtaposed with one-hot and binary encoding in a double log-scale graph. For a very small number of states, adding resilience against multiple faults incurs some overhead penalty, even when compared to one-hot encoding. However, as the state space grows, the complexity of one-hot encoding grows much more rapid with each additional state and eventually becomes infeasible. On the other hand, the relative overhead of EDCs over simple binary encoding decreases monotonically.

Duplication of columns in $\mathbf{G}$ is an easy and efficient way to increase the minimum distance—and, therefore, the resilience—of an EDC. This is helpful against adversarial SMFs such as light attacks. In fact, whenever the code length n exceeds 2^k, $\mathbf{G}$ inevitably contains some duplicated column vectors. Take, for example, the (10, 3) code of resilience four: With $k = 3$, there can only be seven unique nonzero column vectors in $\mathbf{G}$; hence, there must be three duplicates. It turns out that Simplex codes achieve the maximum fault resilience possible without duplication of columns. Their generator matrix consists of all nonzero column vectors of dimension k. They are marked in Table 7 with the small letter s. If $\mathbf{G}$ contains duplicate columns, this means that two or more state variables associated with those columns will have the same value (under fault-free conditions) and could thus be computed by identical next state functions. The inherent problem of identical functions is their susceptibility to CMF, as defined earlier in Section 3.1. If the cause of a fault is not locally limited, it is likely to have the same effect on identical circuit implementations of a Boolean function and, therefore, will increase the fault multiplicity.

6.1 Using Design Diversity to Counter CMF

We can counter the effects of CMF using a technique called *design diversity* [18]. By implementing the next-state function in several different but ultimately equivalent ways, the likelihood of combined failure under the same circumstances is reduced. Design diversity is naturally promoted by defining functions over alternative sets of k state variables, as described in Section 5. It is, however, important to not choose the σ_j naively. For example, if we were to define the next state functions δ_j over only those sets σ_j that all include the state variable s_0, then we would have biased the circuit. A fault in s_0 might spread out to cause subsequent errors in all other state variables. An active adversary could systematically test for such a bias and exploit it. We must therefore balance the use of each state variable across all σ_j. In order to reduce the influence of a single bit error in any of the state variables, the subsets σ_j should uniformly cover all state variables, with minimum overlap. In a k-dimensional state space, the minimum cover implies that each state variable will be used a maximum of k times. The goal is therefore to find n sets σ_j chosen uniformly from $\mathbf{s}_{(n)}$ such that each state variable occurs only k times. Oftentimes, a minimum cover can be found relatively easily, simply by inspection. Automating the process should be fairly straightforward but lies outside of the scope of this paper. Returning to our previous example with the (7, 3)-Simplex code, we could choose the subsets as follows:

TABLE 6
Total Overhead for Modular Exponentiation Circuit with FT Control

	Non-red	3-EDC	4-EDC	TMR	QMR
Data Path	8,396	8,396	8,396	8,396	8,396
Control Logic	2,062	5,942	6,173	8,163	10,773
Total Area	10,458	14,338	14,569	16,559	19,169
Overhead	0.00%	37.09%	39.30%	58.33%	83.29%

TABLE 7
Minimum Code Length n for State Space of Dimension k and Fault Resilience t

t \ k	1	2	3	4	5	6	7
1	2	$^{s}3$	4	5	6	7	8
2	3	5	6	7	9	10	11
3	4	6	$^{s}7$	8	10	11	12
4	5	8	10	11	13	14	15
5	6	9	11	12	14	15	16
6	7	11	13	14	15	17	18
7	8	12	14	$^{s}15$	16	18	19

$$\begin{bmatrix} \sigma_0 \\ \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \sigma_4 \\ \sigma_5 \\ \sigma_6 \end{bmatrix} = \begin{bmatrix} (s_0, s_2, s_3) \\ (s_1, s_3, s_4) \\ (s_2, s_4, s_5) \\ (s_3, s_5, s_6) \\ (s_0, s_4, s_6) \\ (s_0, s_1, s_5) \\ (s_1, s_2, s_6) \end{bmatrix}.$$

One can readily verify that the basis vectors associated with the sets σ_j are all linearly independent and no state variable gets used more than k times. In other words, there is no single state variable that stands out over others as a potential target for an attack.

6.2 Attack Resilience

Finally, we would like to show intuitively how the fault-resilient circuit holds up in different attack scenarios. We start with the most likely point of attack for an adversary with single-mode fault insertion capability, the state register, and continue from there toward the boundaries of the sequential circuit.

- *State register*. A transient fault can completely change the register contents past the duration of the fault since the flip-flops are made up from bistable devices. Here, the structural design of the fault-resilient state machine comes into play: Since the error detection network is connected directly to the output of the state register, all faults up to multiplicity t will be detected. Note that this would not be the case if the detector were to directly follow the next-state function. From our choices of σ_j, one can see that each state variable has an influence on up to k variables of the next state and the attacker would only require a fraction $\lceil t/k \rceil$ of faults to influence more than t state variables. With error detection right after the state register, this threat is eliminated. Furthermore, the detection network does not add to the critical path, thereby avoiding a performance penalty.
- *Feedback path*. Any fault on the feedback path from the state register to the inputs of the next state function would be detected by the error detection circuit as well.
- *Next state logic*. Fault multiplicity stemming from shared logic between individual next state functions is not likely due to the minimum cover policy of selecting the sets σ_j, that is, there should not be any pairs of state variables that are a member of more than one σ_j. However, even if that could happen, it would be easy to simply disallow the sharing of logic gates between individual functions during synthesis.
- *I/O set*. The input and output vectors must also be considered as a potential point of failure, as mentioned earlier. It is not clear at this point if the error detection mechanism of the sequential circuit alone can guarantee t-fault resilience. It might be necessary to have additional error detection outside of the state machine, checking for faults on the I/O set, but this is outside the scope of this paper. The minimum requirement is that the I/O set should contain a sufficiently high amount of redundancy to enable t-fault resilience, as described in the example in the previous section. One potential solution is to use weight-based codes as described in [21].

As we can see from this intuitive analysis, the architecture of our sequential circuit can withstand fault insertion of degree t, which was our design goal.

7 Conclusion

In this paper, we addressed the importance of not only protecting the data path of cryptographic circuits against fault attacks but also the control logic itself. Despite the abundance of fault tolerance techniques for sequential circuits in other application domains, those are based on a fundamentally different fault model and, therefore, not suitable for the task. We strongly believe that these techniques have to be reevaluated and adapted to the new threat scenario that presents itself in the form of cryptographic fault analysis. A determined attacker can probe a system for its weaknesses and enforce a worst-case scenario. For protection against an adversary with the capability of inducing t faults simultaneously, the circuit must be resilient up to that degree. Most of the ideas proposed in the literature, however, are concerned with efficiency and the ability to detect two and correct at most a single error. This work is a first step toward fault-resilient sequential circuits under an adversarial fault model. We gave a motivating example of a hypothetical attack on a modular exponentiation-based cryptosystem that would succumb to this kind of attack in spite of a protected data path and other algorithmic countermeasures. Although it is impossible to ever completely protect against a determined adversary, we presented some general principles of how to improve the resilience against an attack to make it infeasible beyond a certain level of effort.

In this paper, we showed that the overhead incurred through the use of EDCs is less than that of N-modular redundant implementations while providing better fault resilience properties. In addition to this, we demonstrated that the percentage of control logic in a typical cryptographic application is only a small fraction of the data path. Thus, even a high amount of overhead due to fault resilience measures does not significantly influence the complexity of the overall system. It is, however, important to understand the mechanisms of fault propagation to properly implement fault resilience. The rule for minimal overlap between state variables in the sets σ_n is such an example. Furthermore, we defined a new metric that captures the effectiveness of a fault detection method in terms of the minimum effort the attacker has to make to mount a successful attack with respect to the area overhead spent in implementing the error detection scheme. Our analysis and implementation results have shown that the proposed coding-based techniques provide far superior performance when compared against N-modular redundant error detection techniques. For proof of concept, we implemented a Montgomery multiplier whose control logic was realized using various error detection techniques. Our implementation results show that the control logic occupies less than 20 percent of the total area in a nonredundant implementation and the overhead due to fault resilience techniques is limited to around 40 percent for our method, whereas, for a comparable quadruple modular redundancy, it exceeds 80 percent. Furthermore, the effectiveness metric gives a high value of 2.8 for the proposed technique, whereas, for QMR, it is a mere 1.6. At around 75 percent effectiveness after normalization, it is clear that NMR performs even less effectively than the unprotected circuit. For the same implementation, we found that the critical path delay was not affected by the EDC. We provided a rationale to claim that this result would hold for even higher degrees of resilience.

Further research is certainly required in this area. It is our hope that this first step will encourage others to further contribute to this important aspect of embedded cryptography. One open question, for example, is whether other types of codes, for example, Reed-Solomon and so forth, would provide for higher effectiveness. For the time being, we only investigated linear block codes due to the simpler error detection mechanism.

Acknowledgments

The authors would like to thank the anonymous referees for their helpful comments. Gunnar Gaubatz and Berk Sunar were funded in part by the US National Science Foundation Faculty Early Career Development (CAREER) Award NSF-ANI-0133297 and by Intel Corp. Erkay Savaş is supported by the Scientific and Technological Research Council of Turkey Project 105E089.

References

[1] D. Boneh, R. DeMillo, and R. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," *Advances in Cryptology—Proc. EuroCrypt '97,* W. Fumy, ed., pp. 37-51, 1997.

[2] M. Joye, A. Lenstra, and J. Quisquater, "Chinese Remaindering Based Cryptosystem in the Presence of Faults," *J. Cryptology,* vol. 4, no. 12, pp. 241-245, 1999.

[3] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," *Advances in Cryptology—Proc. 17th Ann. Int'l Cryptology Conf. (CRYPTO '97),* B. Kaliski Jr., ed., pp. 513-525, 1997.

[4] G. Piret and J. Quisquater, "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad," *Proc. Fifth Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '03),* C. Walter, Ç.K. Koç, and C. Paar, eds., pp. 77-88, 2003.

[5] H. Choukri and M. Tunstall, "Round Reduction Using Faults," *Proc. Second Int'l Workshop Fault Diagnosis and Tolerance in Cryptography (FDTC '05),* Sept. 2005.

[6] M. Joye and S.-M. Yen, "The Montgomery Powering Ladder," *Proc. Fourth Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '02),* B. Kaliski Jr., Ç.K.. Koç, and C. Paar, eds., pp. 291-302, 2002.

[7] A. Reyhani-Masoleh and M. Hasan, "Towards Fault-Tolerant Cryptographic Computations over Finite Fields," *ACM Trans. Embedded Computing Systems,* vol. 3, pp. 593-613, Aug. 2004.

[8] G. Gaubatz and B. Sunar, "Robust Finite Field Arithmetic for Fault-Tolerant Public-Key Cryptography," *Proc. Second Workshop Fault Diagnosis and Tolerance in Cryptography (FDTC '05),* L. Breveglieri and I. Koren, eds., Sept. 2005.

[9] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "On the Propagation of Faults and Their Detection in a Hardware Implementation of the Advanced Encryption Standard," *Proc. IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors (ASAP '02),* M. Schulte, S. Bhattacharyya, N. Burgess, and R. Schreiber, eds., pp. 303-314, July 2002.

[10] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures," *Proc. Fourth Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '02),* B. Kaliski Jr., Ç. Koç, and C. Paar, eds., pp. 260-275, 2002.

[11] D. Rennels, "Architectures for Fault-Tolerant Spacecraft Computers," *Proc. IEEE,* vol. 66, pp. 1255-1268, Oct. 1978.

[12] M. Chen and E.A. Trachtenberg, "Permutation Codes for the State Assignment of Fault Tolerant Sequential Machines," *Proc. 10th Digital Avionics Systems Conf. (DASC '91),* pp. 85-89, Oct. 1991.

[13] M. Berg, "Fault Tolerant Design Techniques for Asynchronous Single Event Upsets within Synchronous Finite State Machine Architectures," *Proc. Seventh Int'l Military and Aerospace Programmable Logic Devices Conf. (MAPLD '04),* Sept. 2004.

[14] *Fault Tolerant Computing—Theory and Techniques,* D. Pradhan, ed., first ed., vol. 1. Prentice Hall, 1986.

[15] S. Skorobogatov and R. Anderson, "Optical Fault Induction Attacks," *Proc. Fourth Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '02),* B. Kaliski Jr., Ç. K. Koç, and C. Paar, eds., pp. 2-12, Aug. 2002.

[16] R. Anderson and M. Kuhn, "Tamper Resistance—A Cautionary Note," *Proc. Second Usenix Workshop Electronic Commerce,* pp. 1-11, Nov. 1996.

[17] G. Gaubatz, "Versatile Montgomery Multiplier Architectures," master's thesis, Worcester Polytechnic Inst., Worcester, Mass., May 2002.

[18] S. Mitra and E. McCluskey, "Which Concurrent Error Detection Scheme to Choose," *Proc. Int'l Test Conf. (ITC '00),* pp. 985-994, 2000.

[19] A. Hopkins Jr. and T. Smith III, "The Architectural Elements of a Symmetric Fault-Tolerant Multiprocessor," *IEEE Trans. Computers,* vol. 24, no. 5, pp. 498-505, May 1975.

[20] H. Helgert and R. Stinaff, "Minimum-Distance Bounds for Binary Linear Codes," *IEEE Trans. Information Theory,* vol. 19, pp. 344-356, May 1973.

[21] N. Das and N. Touba, "Weight-Based Codes and Their Application to Concurrent Error Detection of Multilevel Circuits," *Proc. 17th VLSI Test Symp. (VTS '99),* 1999.

Gunnar Gaubatz received the BS degree in electrical engineering from the Munich University of Applied Sciences in 2000 and the MS and PhD degrees in electrical and computer engineering from the Worcester Polytechnic Institute in 2002 and 2007, where he was a member of the Cryptography and Information Security Laboratory. He recently joined Intel Corp., Santa Clara, California, as an algorithm development architect. His research interests include fault-tolerant cryptography, computer arithmetic, and low-power digital circuits. He is a member of the IEEE, the IEEE Computer Society, and the International Association for Cryptologic Research (IACR).

Erkay Savaş received the BS and MS degrees in electrical engineering from the Electronics and Communications Engineering Department, Istanbul Technical University, in 1990 and 1994, respectively, and the PhD degree from the Department of Electrical and Computer Engineering (ECE) at Oregon State University in June 2000. He worked for various companies and research institutions before he joined the Faculty of Engineering and Natural Sciences of Sabanci University as a faculty member in 2002. He is the director of the Cryptography and Information Security Group (CISEC) of Sabanci University. His research interests include cryptography, data and communication security, high-performance computing, computer arithmetic, privacy-preserving data mining, and distributed systems. He is a member of the IEEE, the ACM, the IEEE Computer Society, and the International Association of Cryptologic Research (IACR).

Berk Sunar received the BSc degree in electrical and electronics engineering from Middle East Technical University in 1995 and the PhD degree in electrical and computer engineering (ECE) from Oregon State University in December 1998. After briefly working as a member of the research faculty at Oregon State University, he joined Worcester Polytechnic Institute as an assistant professor. Since July 2006, he has been as an associate professor. He currently heads the Cryptography and Information Security Laboratory (CRIS). He received the US National Science Foundation Faculty Early Career Development (CAREER) Award in 2002. He organized the Cryptographic Hardware and Embedded Systems Conference (CHES) in 2004 and is the coeditor of CHES 2005. His research interests include finite fields, elliptic curve cryptography, low-power cryptography, and computer arithmetic. He is a member of the IEEE, the IEEE Computer Society, the ACM, and the International Association of Cryptologic Research (IACR).