Error Detection and Recovery for Transient Faults in Elliptic Curve Cryptosystems

Abdulaziz Alkhoraidly and M. Anwar Hasan
Department of Electrical and Computer Engineering
University of Waterloo

January 20, 2009

Abstract

Faults can corrupt data in storage, in transit, or during a computation. Like other digital systems, cryptosystems are vulnerable to natural and artificial faults. However, the effects of faults on cryptosystems far suppress the corruption of data. Attacks that exploit various classes of faults to learn secret data have been proposed and shown to be practical. As such, efficient detection and recovery of errors resulting from faults have a growing importance in the design of cryptosystems.

We tackle the problem of error detection and recovery for transient faults in elliptic curve scalar multiplication structures. We propose the use of frequent validation with partial re-computation during the scalar multiplication for more efficient error detection and recovery. In our approach, the scalar multiplication iterations are grouped into blocks and efficient error detection schemes are used to detect errors early, which significantly limits the propagation of corrupted data. Moreover, we use the same error detection schemes, combined with partial re-computation, to achieve efficient error recovery without requiring complete time and hardware redundancy. Our analysis illustrates that these modifications enable considerably more efficient and reliable structures relative to known error detection and recovery designs.

1 Introduction

The reliability of digital systems is among the important evaluation criteria, and in some cases the single most important. The existence of undetected faults in medical, military and transportation applications can cause great loss in lives, while undetected faults in financial application can cause great monetary losses. As such, fault tolerance has been among the key design criteria of critical systems, and research has been intensive in developing reliable schemes for error control with minimal effects on performance and cost.

Like other digital systems, cryptosystems are vulnerable to faults of many types. In addition to the data corruption that these faults may cause, attacks have been devised to exploit faults and undermine the security of cryptosystems. Some of these attacks have been demonstrated to be practical. For each of the known fault attacks, countermeasures have been introduced to detect corresponding errors and prevent the resulting information leakage. Moreover, various forms of redundancy have been employed to recover from errors so the system will output the correct result despite the existence of faults.

In general, all computations performed between the occurrence of an error and its detection are corrupted and their results should be discarded. This loss can be reduced by frequent validation of the state of the computation. This may increase the cost in terms of validation tests, but in most conditions, especially in the case of frequent errors or fault-based attacks, this increase is balanced or outweighed by the reduced cost of corrupted computations.

In this work, we propose the use of frequent validation for error detection and recovery in elliptic curve scalar multiplication structures. In our approach, we partition the scalar multiplication iterations into blocks, and use simple and efficient error detection schemes to detect errors early and reduce the loss due to corrupted computations. Moreover, we use error detection schemes combined with partial re-computation to achieve efficient and reliable error recovery without requiring full time and hardware redundancy. Our analysis illustrates that designs employing frequent validation are considerably more efficient and reliable than earlier error detection and recovery designs especially when errors are frequent or in the case of fault attacks.

This document is organized as follows. Section 2 discusses the types and causes of faults in digital systems and addresses known fault attacks on elliptic curve scalar multiplication. It also reviews the reported error detection and fault-tolerant structures for elliptic curve cryptosystems. Sections 3 and 4 discuss the use of frequent validation to achieve efficient error detection and recovery and analyze the performance and reliability advantages. Section 5 provides a summary and discusses future work.

2 Faults, Fault Attacks and Countermeasures

Unlike the direct attacks on the cryptographic algorithms, Side-Channel Attacks (SCA) target the implementation of a cryptosystems by exploiting the information leaking during the proper or improper use of the cryptosystem. An SCA can be passive, where the attacker only observes a working cryptosystem, or active, where the attacker actively attempts to influence the running of the cryptosystem.

Fault attacks on cryptosystems belong to the category of active SCAs and describe a family of attacks where an attacker injects faults into the cryptosystem and uses the resulting faulty results, in addition to the correct result, to discover the secret information partially or fully.

In this section, we give an overview of the causes and types of faults in digital systems and discuss known ways to counter these faults. Then, we examine the known fault attacks on elliptic curve cryptosystems and discuss their countermeasures. Moreover, we review existing error detection and fault-tolerant designs for elliptic curve cryptosystems.

2.1 Faults in Digital Systems

In general, it is important for a digital system to be fault-free and consistently give the correct results. Faults can occur for a variety of natural and artificial reasons, and ways have been proposed to counter their effect on the performance of the system.

Error detection and fault tolerance are even more important for cryptosystems, due to the existence of fault attacks that use faulty results to discover secret information and threat the security of the whole system. In this section, we briefly review the causes and types of faults, and the general hardware and software countermeasures to faults in digital systems.

2.1.1 Causes of Faults

Faults can occur in a device either intentionally or unintentionally, and can be caused by one of many reasons:

1. Variations in standard operation conditions can be used effectively to inject faults in a system. For example:
    (a) The variation in supply voltage can disrupt the execution and cause the processor to skip instructions.
    (b) The variation in the clock frequency, especially for external clocks, can disrupt input/output operations or cause the processor to miss instructions. Moreover,
    (c) Exposing the device to temperatures outside its operational range can cause random modifications of the memory and inconsistencies in memory access.
2. Exposure to light and condensed beams: These methods exploit the photoelectric effects which are inherent in all electric circuits. The exposures to photons induces currents in the circuit that can be used to disrupt the normal operation. Moreover, the targeting and timing can be made more precise by using lasers in fault injection. It is also possible to inject faults in packaged circuits without removing the packaging by using X-rays and ion beams.

2.1.2 Types of Faults

Faults in electronic circuits can either be permanent or transient. Permanent faults are caused by intentional or unintentional defects in the chip. As the name indicates, they have a permanent effect on the behavior of the circuit. Permanent faults can be divided into the following classes:

1. Single event burn-out faults.
2. Single event snap back faults.
3. Single event latch-up faults.
4. Total dose rate faults.

On the other hand, a transient fault does not cause a permanent change in the behavior of the circuit. Such faults are caused by local ionization that induces a current which can be misinterpreted by the circuit as an internal signal. Transient faults can be divided into the following classes:

1. Single event upsets.
2. Multiple event upsets.
3. Dose rate faults.

More information on the different types of faults can be found in [5].

2.1.3 Statistical Modeling of Faults

Modeling the occurrence of faults in a system is important in estimating their frequency and effects, and hence, in minimizing the cost incurred by error detection and fault tolerance schemes by designing them accordingly. Reliability is usually modeled for single components in a system, and then the results are used to estimate the reliability of the whole system.

An important parameter that describes the reliability of a component is its *failure rate*, denoted by λ, which describes the expected number of failures per unit time for a component in a good condition [8]. In general, components go through a period of high failure rate early in their lifetime, the *infant mortality* phase. Then, their failure rate stabilizes for a considerable amount of time in their *operational phase.* Towards the end of their operational lifetime, components usually display increasing failure rates, the *wear-out* phase.

A model for reliability [8] Let T be a random variable that describes the time until a component fails and let $f(t)$ and $F(t)$ denote the probability density function of T and its cumulative distribution function, respectively. Then, the reliability of a component, denoted by $R(t)$, is defined as the probability that the component will not fail before time t, i.e. $R(t) = \Pr(T > t)$.

It can be shown that, for a component in its operational phase, i.e. with a constant λ, the lifetime of the component will follow an *exponential distribution* with parameter λ, so for $t \geq 0$, we have $f(t) = \lambda e^{-\lambda t}$, $F(t) = 1 - e^{-\lambda t}$, and $R(t) = e^{-\lambda t}$. Moreover, the mean time to failure (MTTF) is the expected value of T and is equal to $\frac{1}{\lambda}$. Note that the exponential distribution is a *memoryless* distribution, which means that for any real numbers t_0 and t_1, $\Pr(T > t_1 | T > t_0) = \Pr(T > t_1 - t_0)$, i.e. the distribution has no memory of earlier samples. In other words, time frames of equal length have the same expected number of faults regardless of their position relative to $t = 0$.

When components are combined to make up a system, their individual reliabilities can be used to estimate the reliability of the whole system based on the dependency relationships between the components. It is usually assumed that components fail independently from each other [10], so the reliability of the whole system can be derived using the probability rules for independent events.

2.1.4 Fault-tolerant Designs

Several solutions have been devised to avoid or detect faults, or detect the attempt to inject them. Other solutions help to tolerate the occurrence of faults and produce correct results in spite of their existence. Some of these methods are implemented in hardware while some can be implemented in software or as a combination of the two.

The main countermeasure against faults and errors is the use of *redundancy* in the design. A redundant design makes it possible to detect faulty results and behaviors. It also may permit the recovery from faults.

The simplest form of redundancy is *hardware redundancy*, which entails replicating some part of the hardware to prevent the existence of a single point of failure. Results produced by different modules are compared to detect faulty operation and, in certain cases, produce the correct result.

1. Static redundancy: In this type of hardware redundancy, one or more replica of the module are run in parallel, and their results are compared to detect the existence of faults, and possibly to confirm the right result by a majority vote. Some examples of this type of redundancy are:

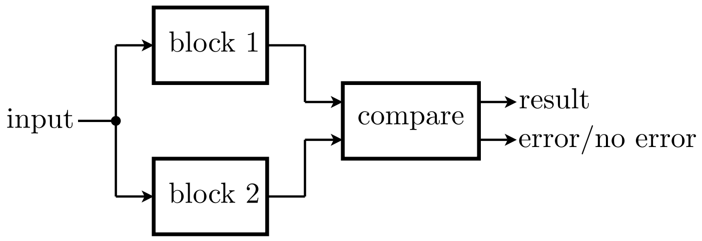

Figure 1: Double modular redundant structure

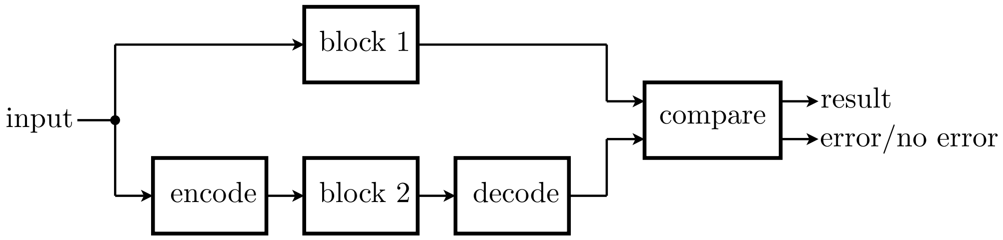

Figure 2: Double modular redundant structure with input coding

(a) Simple duplication: An example of simple duplication is double modular redundancy (DMR) illustrated in Figure 1. DMR allows for detection of transient and permanent faults provided that faults that occur in the two modules generate different results. However, when a fault occurs, this setup does not generally help in tolerating faults unless there is a way to check whether a given result is the correct one.

A variant of this scheme involves encoding the input to one of the modules or both of them in a way that allows getting the intended result by decoding the output of that module, as illustrated in Figure 2. When this is applied, similar faults will generate different results, which helps in detecting them. The applicable encoding methods depend on the characteristics of the computation and may include negation, swapping, shifting and randomization [5].

(b) Multiple duplication: A common example of multiple redundancy is triple modular redundancy (TMR) illustrated in Figure 3. The main advantage of TMR compared to DMR is the possibility of tolerating faults through majority vote. A more general case is the N-modular redundancy (NMR), shown in Figure 4, which gives higher reliability but has a much higher cost. As before, it is possible to employ input encoding to get a better error detection capability.

2. Dynamic redundancy: This type of redundancy is similar to the multiple redundancy discussed earlier with the main difference that the voting block is replaced with a switch that is controlled by an error detection block, as illustrated in Figure 5. With exception of one of the modules, designated as the main module, other replicas may be working in parallel or turned off and used as spares when the main module fails.

3. Hybrid redundancy: Hardware redundancy can also be implemented as a hybrid of static and

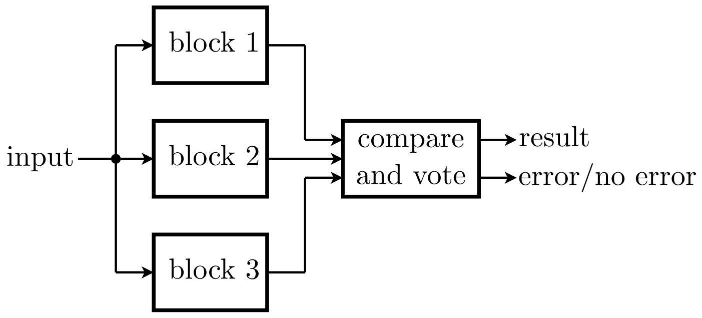

Figure 3: Triple modular redundant structure

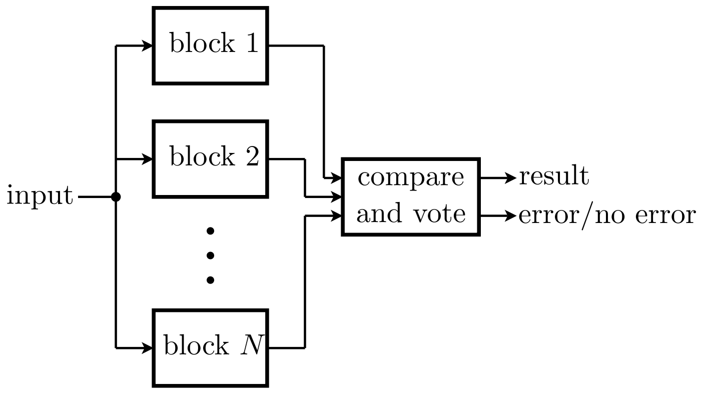

Figure 4: N-modular redundant structure

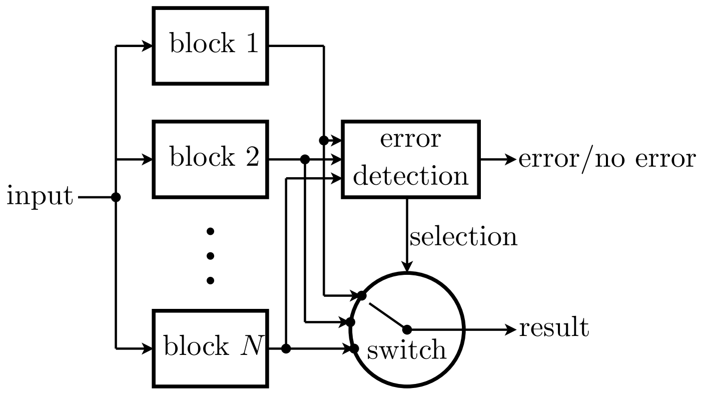

Figure 5: Dynamic redundancy

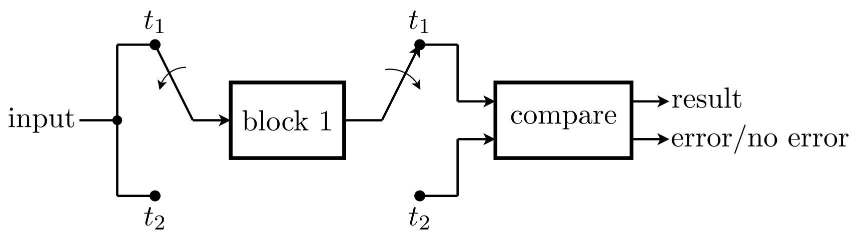

Figure 6: Time redundancy with input coding

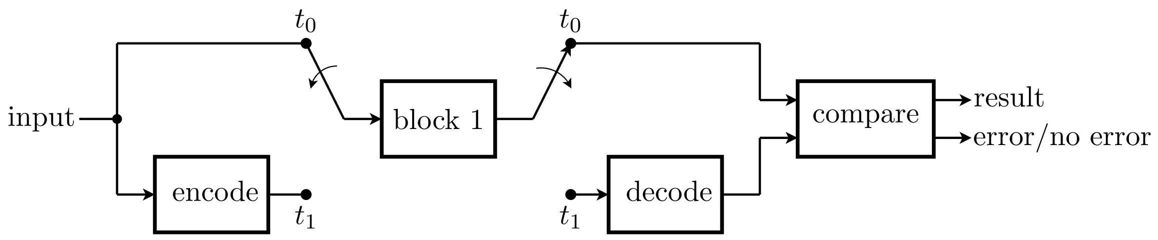

Figure 7: Time redundancy with input coding

dynamic redundancy, with the possibility of using input encoding for added error detection capability.

Another form of redundancy is *time redundancy*, which means repeating the computation or a part of it to confirm the earlier results and detect transient faults. Note that pure time redundancy is not effective to detect permanent faults that produce consistent errors.

1. Simple time redundancy: As illustrated in Figure 6, simple time redundancy involves the repeating the operation using the same hardware module to detect any difference in the results that may indicate the occurrence of a fault. This scheme can be made more sensitive to faults by input encoding, as shown in Figure 7 due to the difficulty of injecting faults that have the same effects on two different computations. The applicable encoding methods depend on the characteristics of the computation and may include negation, swapping, shifting and randomization [5]. However, like in the case of DMR, this setup does not generally help in tolerating faults unless there is a way to check whether a given result is the correct one.

2. Multiple time redundancy: As an extension to simple time redundancy, multiple time redundancy, illustrated in Figure 8, involves repeating the operations more that twice. This has the potential of detecting errors and possibly correcting them. Also, input encoding can be employed for more sensitive error detection.

A third form of redundancy is *information redundancy*, which is commonly employed in data communication through error control codes. The principle behind information redundancy is the use of more bits to represent the data than is actually necessary. This way, some of the representable bit patterns don't correspond to valid data and can be used to detect and correct errors. Some examples of this type of redundancy are checksums and error detection and correction codes.

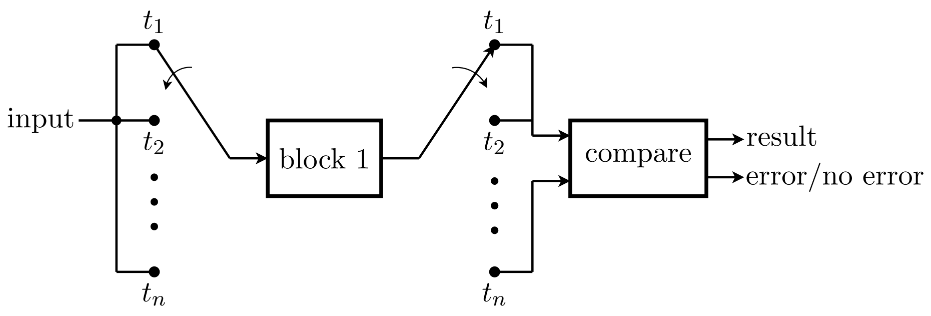

Figure 8: Multiple time redundancy

It is also possible to combine two or more types of redundancy in a single scheme to get the advantages of different types of redundancy. One good example is error detection and fault tolerance solutions for elliptic curve scalar multiplication where it is possible to combine all types of redundancy, as we will discuss later.

Other countermeasures to fault injections that are commonly employed include:

1. Sensors and detectors: A variability in the operating environment can be detected using suitable sensors. For example, a voltage detector can be used to ensure that the supply voltage is within the operating range of the circuit.

2. Metal shields: Metal meshes can be used to cover the circuit, and can be either active or passive. An active metal shield has data continuously passing in it to prevent probing and to prevent exposing the circuit. A passive metal shield, on the other hand, doesn't have any data passing in it, but it helps by shielding the circuit and reducing the outgoing radiation.

3. Encryption: The communication between the processor and the memory, as well as the memory contents and addressing, can be encrypted with changing keys to make it more difficult to target a certain memory location.

4. Unstable frequency generators: The use of unstable frequency generators makes it difficult for the attacker to synchronize with the device as every run will have different timings.

5. Software baits: Small code fragments (baits) can be inserted into the software running on the device to perform simple operations and validate the results to check the functional correctness of the device.

6. Randomization: Injecting randomness into either the data to be processed or the order of execution makes it difficult for an attacker to target a certain instruction or variable. Moreover, some forms of data randomization don't prevent the attack but rather mask the faulty results so that they cannot be used by the attacker.

2.2 Fault Attacks on Elliptic Curve Cryptosystems

In general, fault tolerance is important for the correct functioning of all systems. However, it is more essential for cryptosystems to take care of faults since their effect may go beyond interrupting the functionality to threaten the security of the system by manipulating it to leak some of its secret information. In this section, we discuss some of the known fault attacks that target elliptic curve cryptosystems and their countermeasures.

2.2.1 Biehl-Meyer-Müller Invalid Curve Attacks

In the attacks presented in [6], the representation of a point P on a strong elliptic curve E can be modified, e.g. by a register fault, to move P to a different, often weaker, curve E'. The resulting incorrect output values can be used to find possible intermediate values in the computation, which reveals parts of the secret key. Usually, the attack has to be repeated since in many cases the value guessed are not unique.

Basic attack Let E be a strong elliptic curve defined over a finite field K as

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

and let P and $Q = kP$ be two points on E. To be able to mount this attack, suppose that the device doesn't check whether P and Q are actually on E.

According to the ANSI X9.63 and IEEE 1363 standards, a_6 isn't used in the addition operation. It follows that for a point $P' = (x', y')$ with $x', y' \in K$ and $P' \notin E$, the calculation of $Q' = kP'$ occurs over the curve $E'(a_1, a_2, a_3, a_4, a_6')$ where

$$a_6' = y'^2 + a_1x'y' + a_3y' - x'^3 - a_2x'^2 - a_4x'$$

instead of the original curve E. If P' is chosen in such a way that E' is a cryptographically weak curve, i.e. P' has a relatively small order a over E', then the value of k modulo a can be found be solving a DLP over the subgroup of order a generated by P'. This process can be repeated until sufficient residues of k are collected, and then CRT is used to calculate k. This attack runs in polynomial time.

Exploiting early random register faults Now suppose that the device checks whether P lies on E before starting the computation, and assume a single bit fault in an unknown position can be injected right between the test and the computation. Again, the modified point P' will lie on a curve E' with different, yet unknown, a_6' value.

Using the incorrect output $Q' = kP'$, the value of a_6' can be computed. For each of the possible values of P' that lie on E', we find k' such that $Q' = k'P'$ by solving a DLP on E'. Then, we proceed to compute k as in the basic attack. This attack can be used to attack both the El Gamal encryption and the ECDSA protocol in a sub-exponential running time.

Exploiting random faults during computation Assume that E is defined over an extension field $\mathbb{F}_q$ such that $E(\mathbb{F}_q)$ contains a subgroup of prime order p with $p > q/\log q$, and that the binary algorithm is used to perform the scalar multiplication. Also, assume that the attacker can repeatedly input a point P and induce a register fault during m successive iterations of computation, and that the correct result $Q = kP$ is known. Let Q_i denote the values of Q before the i-th iteration.

During the computation, a fault is injected in a random iteration j to get Q_j' and eventually Q'. Then, the values of Q, Q' ,Q_j and Q_j' can be used by successive guessing and refinement to discover the iteration j and an intermediate value Q_j', which can be used to guess the higher $n - j$

bits of k. Then the process can be repeated going downwards through the bits of k by inducing new random faults in blocks of at most m consecutive iterations. The choice of the value of m presents a trade-off between the number of register faults required and the time needed to analyze the faulty results.

2.2.2 Ciet-Joye Invalid Curve Attacks

In the attacks presented in [9], the assumption that only a single or few bit errors can be injected into the representation of P is relaxed. It also demonstrates how random errors in either the representation of P, the curve parameters or the field representation can allow for the recovery of secret key either fully or partially.

Faults in base point Let E be an elliptic curve over K and let $P = (x_P, y)$ be the base point. Faults can be assumed to occur in either or both of the coordinates of P. For example, assume that the value stored for the x-coordinate, x', is corrupted in some unknown bit positions and let $P' = (x', y)$. Then, $Q' = kP'$ can be computed and will lie as before on the curve E', which shares all the parameters of E except a_6. The corresponding value for E', a'_6, can be computed from the coordinates of Q' as

$$a'_6 = y_{Q'}^2 + a_1 x_{Q'} y_{Q'} + a_3 y_{Q'} - x_{Q'}^3 - a_2 x_{Q'}^2 - a_4 x_{Q'}$$

Then, we know that x' is a root in K of

$$x^3 + a_2 x^2 + (a_4 - a_1 y)x + (a'_6 - y^2 - a_3 y)$$

since $P' \in E'$.

Assuming that r, the order of P' in E', is small, the root of the above polynomial with the least Hamming distance from the original x_P can be used as a candidate for x'. Assuming that the DLP on E' is weak, the Pohlig-Hellman reduction can be used to obtain a candidate for k.

Faults in system parameters Faults can also be injected in the field parameters, either in storage or in transit. Let E be a curve defined over a prime field $\mathbb{F}_p$ of characteristics other than 2 and 3. In such a case, the equation of E can be simplified to $y^2 = x^3 + ax + b$. Assume that a bit error is injected in p to give the almost similar value p' and that all field operations will then be performed modulo p' instead. In particular, the values of P, Q, a and b will be represented modulo p' as P', Q', a' and b', respectively.

Since Q' satisfies the equation of E', it follows that

$$b' \equiv {y'}_Q^2 - {x'}_Q^3 - a' \equiv b \equiv y^2 - x^3 - a \mod p'$$

Hence, $p'|D$ where $D = \left| {y'}_Q^2 - {x'}_Q^3 - a' - (y^2 - x^3 - a) \right|$ and p' can be revealed through factoring D as the product of factors that has the shortest Hamming distance from p. Using these factors, the value of k can be computed by solving a set of small DLPs and using the CRT. Faults in binary fields representations and in other system parameters can be exploited in a similar way.

2.2.3 Sign-change Fault Attack

Earlier fault attacks on ECC worked by inducing faults in a way that would move the computation to a different (and probably weaker) elliptic curve. This can be achieved by either changing the base point, an intermediate point, or a parameter of the curve. However, this can be easily countered by checking that the result belongs to the original curve.

The attack described in [7] doesn't move the computation to a different curve, but rather results in a faulty point on the original curve. By collecting enough of these faulty results, the secret can be recovered in expected polynomial time.

It follows that point checking cannot be used to counter such attacks. Actually, it may help the attacker to remove useless faulty points, i.e. points that fall off the curve, and hence it makes a less precise attack more effective. Randomization can be used as a countermeasure, but it is not an acceptable solution in some implementation standards. However, the Montgomery scalar multiplication that doesn't use the y-coordinate is immune to such attack.

Attack description Basically, the attack involves changing the sign of an intermediate point in the scalar multiplication algorithm. The sign change can happen uniformly in any iteration, and the attacker doesn't know the precise iteration in which the change happened.

The attack can be mounted on different intermediate variables. For example, in a left-to-right scalar multiplication algorithm, the attack can be applied to the doubling step, i.e. $Q_i = 2Q_{i+1}$. If the sign of Q_i is changed, the final result would be $\tilde{Q} = -Q + 2L_i(k)$, where Q is the correct result and $L_i(k)$ is the product of P and the lowest i bits of k.

The idea is to recover the bits of k in blocks of $1 \leq r \leq m$ bits, where m is chosen to control a trade-off between the required number of faulty results to achieve 1/2 chance of success ($n/m \log 2n$), and the amount of offline search (2^m).

The attack starts by collecting $n/m \log 2n$ faulty points. Then, starting from the least significant bit, the attacker attempts to inductively guess the bits of k by trying to find a bit pattern of size $r \leq m$ and a faulty point $\tilde{Q}$ such that $\tilde{Q} = -Q + 2L_i(k)$. If a faulty point is found to work, then the corresponding bit pattern is considered a good guess. Otherwise, it assumes that the least bit is 0 and continues. Lastly, it checks the final guess by computing kP, which should succeed with a probability at least 1/2.

2.2.4 An Overview of Countermeasures for Fault Attacks

As discussed earlier, fault attacks on elliptic curve cryptosystems can be divided into two classes, namely, attacks using an invalid curve and attacks on the valid curve. In the first class, there are the attacks by Biehl, Meyer and Müller [6], attacks by Ciet and Joye [9], and their variants. In the second class, the only known attack to date is the sign-change attack by Blömer, Otto and Seifert [7]. Here, we briefly discuss the known countermeasures for these attacks and comment on their effectiveness and limitations.

1. Checksums: The use of checksums, like in error-correcting RAMs, helps in detecting errors in data while stored or in transit, but can not be used to detect errors injected during the computation. As such, they are not enough as a countermeasure against neither invalid-curve attacks nor sign-change attacks.

Algorithm 1 Right-to-left double-and-add-always SM algorithm with point validation and consistency checking [3]

```
Input: $P \in E(K)$, $l = (l_{n-1}, l_{n-2}, l_{n-3}, \dots, l_0)$
Output: $lP$
1: $Q_0 \leftarrow \mathcal{O}$, $Q_1 \leftarrow \mathcal{O}$, $Q_2 \leftarrow P$,
2: for $i = 0$ to $n-1$ do
3:     $Q_{l_i} \leftarrow Q_{l_i} + Q_2$
4:     $Q_2 \leftarrow 2Q_2$
5: end for
6: if $Q_0 \in E(K)$ and $Q_1 \in E(K)$ and $Q_2 = Q_0 + Q_1 + P$ then
7:     return $Q_1$
8: else
9:     return $\mathcal{O}$
10: end if
```

2. Hardware and/or time redundancy: With redundancy in time or hardware, accompanied by comparison, it is easier to detect faults since the attacker has to inject the same error twice to pass the comparison test. In particular, hardware redundancy helps also in detecting permanent faults in the computation block or the system parameters. The main disadvantage of this solution is its cost, either in time or in hardware resources.

3. Point validation: The representation of elliptic curve points has some inherent information redundancy that can be used to detect invalid points, i.e. points that do not belong to the used curve. It follows that validating the input and output points can defeat invalid-curve attacks. Moreover, point validation is a relatively cheap operation that requires only applying the curve equation to the coordinates of the point to be checked. However, since the sign-change attack does not attempt to move the point to an invalid curve, it cannot be countered by point validation alone.

4. Scalar multiplication using Montgomery's ladder: Since the Montgomery scalar multiplication algorithm works using only the x-coordinates of points, it is naturally immune to the sign-change fault attack. It is also immune to invalid curve attacks that use the y-coordinate. However, Montgomery's ladder is not enough by itself to counter a general invalid curve attack that does not use the y-coordinate.

5. Randomized encoding: Randomization can be used in elliptic curve scalar multiplication in a variety of ways while encoding the scalar, base point or curve parameters. For example, splitting of the scalar k or adding a random point to the base point P can be used to mask the faulty result of a sign-change fault attack, preventing the attacker from using it. However, randomization does not help in countering invalid-curve attacks since its effect will be reversed by the decoding of the result.

6. Computation on a combined curve: This countermeasure, proposed in [7], combines time and information redundancy, and is similar in principle to the use of a residue number system to counter fault attacks, or to Shamir's method to protect RSA systems [1]. Basically, a smaller curve is chosen and the scalar multiplication is performed twice, once on the combined curve and once on the smaller curve. Computation on a combined curve is an effective

countermeasure against sign-change fault attacks, but it can be shown that it is not as effective against invalid-curve attacks [7]. Moreover, it requires significantly more time than consistency checking.

7. Consistency checking: Some algorithms involve built-in redundancy that allows for checking the consistency of the results. For example, in Montgomery's algorithm, the point Q and H always satisfy $Q = H + P$. Another example is Algorithm 1, which is a right-to-left double-and-add-always algorithm in which intermediate variables satisfy an invariant that can be used to check for consistency. This invariant, combined with point validation, gives resistance to both invalid-curve and sign-change fault attacks [3].

2.3 Fault Tolerance in Elliptic Curve Cryptosystems

Section 2.1.4 has outlined generic schemes for fault tolerance while Section 2.2.4 summarized the known countermeasures specific to elliptic curve cryptosystems. In this section, we discuss designs that use some of the known solutions and countermeasures to come up with fault-tolerant elliptic curve scalar multiplication structures. We will look at traditional and recently proposed designs [2] and evaluate them in terms of time and hardware overhead, probability of indicted errors, and reliability.

2.3.1 Error Detection in Elliptic Curve Cryptosystems

In the context of elliptic curve cryptosystems, error detection involves the detection of errors resulting from both natural and artificial faults, whether transient or permanent, with a very low probability of undetected errors. As discussed earlier, natural errors and most errors caused by an attacker will move a point to an invalid curve, with the exception being errors caused by a sign-change attack where the point stays on the original curve.

Known error detection solutions for elliptic curves employ information redundancy, hardware redundancy, time redundancy or a combination thereof.

Information redundancy One of the simplest solutions, and the least expensive in terms of overhead, is validating the resulting point before giving it as an output. Point validation (PV) can be performed using the same field arithmetic units used in the scalar multiplication operation, or can be performed by an independent module, and takes around 80-200% of the time of an average point operation depending on the field and the implementation. PV does not protect against sign-change attacks. For invalid curve attacks, $\Pr(\text{undetected error})_{\text{PV}} \approx \frac{1}{q}$, where q is the order of the underlying field [2]. A more general approach, namely consistency checking (CC), can be employed against all known types of errors resulting from fault attacks [2].

Hardware redundancy Another solution for error detection is simple hardware redundancy, i.e. parallel computation (PC) with comparison. A straight forward DMR implementation of elliptic curve scalar multiplication will detect an error when the resulting points from the two modules are different, e.g. caused by a permanent fault in one of the modules. However, for a sophisticated attacker, it would be possible to inject the same fault in both modules to get the same errors at the output. In this case, the structure will accept the corrupt result as valid.

An enhancement for this design using input encoding has been proposed in [2]. There are many ways to encode the inputs of a scalar multiplication based on the properties of elliptic curves, which enables the computation to be effectively randomized. In [2], two encoding methods have been chosen, namely, point randomization in projective coordinates and adding multiples of the curve order to the scalar. These methods have been chosen since they require no output decoding and due to their lower probability of undetected errors in experiments.

When input decoding is used, the computation is randomized and it is much more difficult for the attacker to inject faults in a way that causes the two modules to output the same faulty result. As such, this structure can detect errors caused by natural faults, both transient and permanent, and attacker-induced faults, even sign-change faults. The probability of undetected errors is much lower for this structure, namely $\Pr(\text{undetected error})_{\text{PC}} \approx \frac{1}{q^2}$. While this structure takes almost the same time as one scalar multiplication, its space requirements are doubled.

Time redundancy Re-computation is a straight forward application of time redundancy with comparison. As before, simple re-computation (RC) can detect errors caused by transient faults only when the two results are different, and can not detect errors caused by permanent faults. An advanced attacker can inject the same fault in the two runs and make the structure output a faulty result.

Input encoding has also been applied to re-computation [2]. In this case, the structure will be able to detect both transient and permanent natural errors due to the randomization of the computation and since the same fault will have different results. Also, it would be very difficult for the attacker to inject faults in a way that make the outputs both equal and faulty. As such, this structure has the same probability of undetected errors as the parallel computation design, i.e. $\Pr(\text{undetected error})_{\text{RC}} \approx \frac{1}{q^2}$. The hardware requirements of re-computation with input encoding are the same as the conventional scalar multiplication, but the running time in this case is doubled.

A variant of this solution is proposed in [2] where, instead of full re-computation, only partial re-computation is required. In this case, the result of the partial computation is compared to an intermediate result in the complete one. Probability of undetected errors depends on fault type and position but is generally low unless the attacker can specifically inject faults beyond the segment that is used for comparison.

2.3.2 Fault Tolerance for Elliptic Curve Cryptosystems

The use of error detection schemes is enough to protect a cryptosystem from both natural faults and fault attacks. However, it may be beneficial in some scenarios to generate the correct result even in the presence of a fault. In this section, we review traditional and recent fault-tolerant structures for elliptic curve scalar multiplication.

Triple modular redundancy (TMR) Triple modular redundancy with a majority vote works well when faults are limited to one block, while when two or more blocks produce different faulty results, it can detect faults but cannot determine the correct result. However, an advanced attacker can inject the same fault in two or more modules and make the structure output a faulty result.

As before, this has been solved in [2] with input encoding. The inputs of each of the modules are encoded differently and the results are compared. In this case, the computations will be randomized and similar faults will generate different faulty results. The probability of undetected errors for this structure is low, $\Pr(\text{undetected error})_{\text{TMR}} \approx \frac{3}{q^2}$, its running time is almost the same as a scalar multiplication, but the space requirements are tripled. Moreover, the reliability of a TMR structure is $R_{\mathrm{TMR}} = 3R_{\mathrm{ECSM}}^2 - 2R_{\mathrm{ECSM}}^3$, where R_{ECSM} is the reliability of a stand-alone scalar multiplication unit.

Dual modular redundancy with point validation (DMR-PV) [2] As the name indicates, this is a simple replication with comparison scheme combined with point validation. The inputs to each of the modules are encoded as before, and the results of each are tested by a point validation module.

This structure can detect both natural-cause errors and errors caused by an invalid-curve fault attack limited to one module, and detect them when they occur in both modules. However, this structure can not always detect errors caused by sign-change faults since an attacker can inject a sign-change fault in one of the modules and a random fault in the other, and in this case, the structure will detect the error caused by the random fault and output the other faulty result as the correct result.

For errors not caused by a sign-change fault attack, this structure has a relatively low probability of undetected errors, $\Pr(\text{undetected error})_{\mathrm{DMR-PV}} \approx \frac{2}{q}$. It requires similar time to a scalar multiplication, but its hardware requirements are doubled. Moreover, the reliability of a DMR-PV structure is $R_{\mathrm{DMR-PV}} = 2R_{\mathrm{ECSM}} - R_{\mathrm{ECSM}}^2$.

Parallel computation and re-computation (PRC) [2] As before, this is a combination of two types of redundancy, namely hardware redundancy and time redundancy with input encoding. Two modules are used and their inputs are encoded, then their results are compared. If the results are equal, one of them is given as the correct result. Otherwise, the computation is performed again with different input encoding for both modules and the new results are compared with the old ones to find the correct result.

This structure can recover from all errors limited to one of the modules and detect them when they occur in both modules. Moreover, due to the randomized computations, it is very difficult for an attacker to inject faults in a way that generates equal faulty results. As such, this structure is effective against both invalid-curve and sign-change fault attacks.

This structure has a very low probability of undetected errors, $\Pr(\text{undetected error})_{\mathrm{PRC}} \approx \frac{3}{q^2}$. Its space requirements are double those of a scalar multiplication, and its time requirements can be the same when no errors are detected, and doubled otherwise. Moreover, the reliability of a PRC structure is similar to the DMR-PV structure, namely, $R_{\mathrm{PRC}} = 2R_{\mathrm{ECSM}} - R_{\mathrm{ECSM}}^2$.

3 Frequent Validation for Efficient Error Detection in ECC

As discussed in the previous section, error detection and fault tolerance for elliptic curve scalar multiplication operation have been usually achieved by testing data for validity before and after execution and randomizing the encoding of inputs, essentially treating the scalar multiplication operation as a black-box. While this approach has its merits, particularly in that it minimizes the modifications to existing scalar multiplication structures and the number of tests required, it has the disadvantage of allowing an error to propagate corrupt ting all the following iterations until the end of the scalar multiplication operation.

In this section, we propose the use of frequent validation as a more efficient way to achieve error detection in elliptic curve scalar multiplication. Instead of a single validity test at the end of the computation, intermediate results are tested so errors can be detected early. This modification has advantages in terms of both time and hardware requirements and the reliability of resulting structures, as will be discussed later.

3.1 Fault Model and Assumptions

In this work, we deal with errors caused by transient faults that affect iterative algorithms. In particular, we assume that the iterative algorithm at hand satisfies two conditions:

1. The state of the algorithm, i.e. intermediate results, can be tested for validity.
2. The validity tests are relatively efficient.

Faults considered here, whether induced naturally or by an attacker, are assumed to be transient. Also, when a fault occurs anywhere in an iteration of the algorithm, we consider the whole iteration faulty. Generally, an error caused by a fault will propagate to later iterations and the final result will be faulty. We assume that repeating the faulty iteration with the absence of the transient fault would produce a correct result. An iteration can be either faulty or not, and we assume that all iterations have the same probability of being faulty, p. In this context, frequent validation seeks to discover errors in the computation as early as possible. When the goal is error detection, early error discovery will prevent the waste (in time and power) of proceeding with the computation after an error has occurred. When the goal is error recovery, it would be possible to recover from errors by re-computing the faulty iterations. This way, we avoid repeating the whole computation, including the non-faulty part.

Each iteration can be modeled as a Bernoulli experiment with a probability of being faulty p. Let n denote the total number of iterations. For frequent validation, we divide the computations into blocks of m iterations, and test the validity of the results after each block is computed. As in the conventional reliability model, we assume that faults are statistically independent among iterations. Then, the number of non-faulty iterations before observing the first error, denoted by X, will follow a geometric distribution. The probability of m consecutive iterations without an error (i.e. of a successful block) can then be given by $\Pr(X \geq m) = (1-p)^m$, while the probability of an error or more in a block is the complement $\Pr(X < m) = 1 - (1-p)^m$.

Let $c(x)$ be the cost of computation in point operations for x iterations. (For the conventional double-and-add algorithm, $c(x) = 3x/2$ point operations on average, while for 2-NAF double-and-add algorithm, $c(x) = 4x/3$ on average.) Let c_v be the cost of the test or the validation check that can be used to detect the occurrence of a certain class of faults.

Relationship to the conventional reliability model The model we adopt here is related to the conventional reliability model described in Section 2.1.3. In particular, a geometric distribution with parameter p is the discrete analogue of an exponential distribution with parameter $\lambda = \ln\left(\frac{1}{1-p}\right)$ [4]. Based on this relationship, Figure 9 illustrates the reliability associated with different iteration fault probabilities for an operation with 163 iterations, and clearly shows that the practical region of iteration fault probabilities is $p < 0.05$.

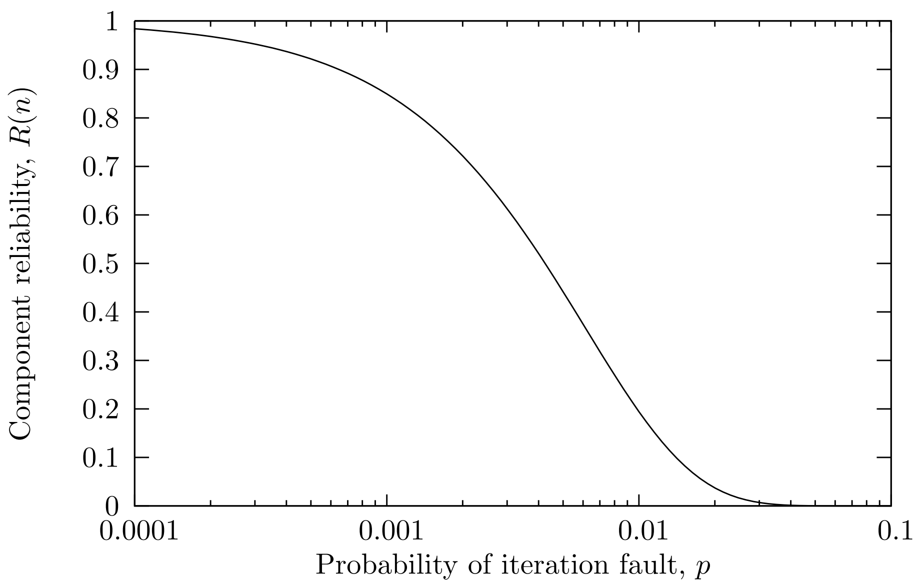

Figure 9: Component reliability vs. iteration fault probability for $n = 163$ iterations

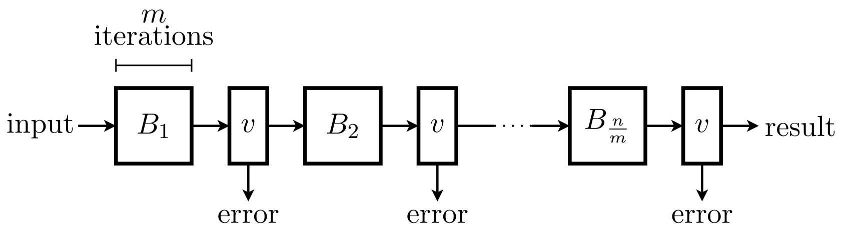

Figure 10: Error Detection by Frequent Validation

3.2 Error Detection with Frequent Validation

Error detection is generally achieved using a test at the end of the computation which is designed to detect a certain type of errors, and these tests are usually efficient. However, when an error occurs, all computations performed after the occurrence of the error will be affected, and performing them becomes wasteful. This is significant in terms of time, and in case of mobile devices, power consumption.

So, it is important to design error detection structures in a way that is not only efficient, but also limits the wasted computations after the occurrence of an error. Our approach to solve this problem is the use of frequent validation. The idea is to prevent error propagation using multiple validation tests of intermediate results rather than a single test at the end, as illustrated in Figure 10. While this approach may slightly increase the cost of testing, this cost will be offset by the saving in iterations lost due to error propagation.

3.2.1 A Model for Error Detection with Frequent Validation

As mentioned earlier, frequent validation involves dividing iterations into blocks and performing a validity test at the end of each block. Then, if only error detection is sought, the computation can be terminated when an error is detected. In this section, we analyze the cost of the increased testing and the gain of preventing the execution of faulty computation following an error. We will also show that the block size can be chosen to minimize the expected cost of wasteful computations.

Recall that n is the total number of iterations, m is the block size in iterations, p is the probability of an error in an iteration, and that iterations are assumed to be statistically independent. It follows that the probability of having m consecutive successful iterations is $\Pr(X \geq m) = (1-p)^m$, so the probability of a faulty block (with one or more errors) is $\Pr(X < m) = 1 - (1-p)^m$.

The random variable describing the number of non-faulty blocks needed to observe a faulty one, denoted by Y, follows also a geometric distribution with mean $E(Y) = 1/(1-(1-p)^m)$. However, this expression of the mean takes into account an infinite number of potential experiments, which is not the case in our scenario since we have a finite number of blocks, namely $\frac{n}{m}$. Whenever an error occurs after the $\frac{n}{m}$ blocks, the whole computation is considered as a success rather than a failure, and since the mass of the distribution beyond $\frac{n}{m}$ is not considered for mean calculations, the mean in this scenario would obviously be less than in the infinite case.

To find the expected value for a finite number of experiments, $E_{\frac{n}{m}}(Y)$, which would be less than the expected value for the infinite case, $1/(1-(1-p)^m)$, we find the weighted average of all the relevant outcomes, i.e. 1 through $\frac{n}{m}$, and normalize it by the cumulative distribution function (CDF) of Y up to $\frac{n}{m}$ trails to get

$$E_{\frac{n}{m}}(Y) = \frac{\sum_{i=1}^{\frac{n}{m}} i \Pr(Y = i)}{\Pr(Y \leq \frac{n}{m})}$$

where $\Pr(Y \leq \frac{n}{m}) = 1 - ((1-p)^m)^{\frac{n}{m}} = 1 - (1-p)^n$.

With this expected value, we can now analyze the cost and benefit of the frequent validation approach to error detection. By checking early for an error, our aim is to reduce the computation performed after the error while adding the least overhead possible due to extra tests. There are two cases to be considered: (i) no error or an error after the $\frac{n}{m}$ blocks, or (ii) an error within the $\frac{n}{m}$ blocks. In each case we will estimate the cost of testing and the wasteful computation due to an error.

In the first case, namely no error or an error after $\frac{n}{m}$ blocks, the number of tests would be equal to the number of blocks, $\frac{n}{m}$, so their cost would be $c_v \frac{n}{m}$. Also, there would be no waste since there was no error in the $\frac{n}{m}$ blocks. The probability of this case is $\Pr(Y > \frac{n}{m}) = (1-p)^n$. The contribution of this case would then be $\Pr(Y > \frac{n}{m}) c_v \frac{n}{m}$

In the second case, namely that an error happened within the $\frac{n}{m}$ blocks, we observe that a test is needed for each block until one is found faulty. So, the expected number of tests is equal to the expected number of blocks needed to observe the first error, $E_{\frac{n}{m}}(Y)$. The expected cost of the tests is then $E_{\frac{n}{m}}(Y) c_v$.

To estimate the wasted computation due to an error, we consider the faulty iteration and all the following iterations until the error is detected, i.e. until the end of a block. In a block, the number of iterations needed to observe a faulty one, X, follows a geometric distribution with an expected value of $1/p$. However, for the same reasons discussed earlier, the fact that the block has a finite number of iterations reduces the expected value to

$$E_m(X) = \frac{\sum_{k=1}^{m} k \Pr(X = k)}{\Pr(X \leq m)}$$

Figure 11 illustrates this for $n = 163$ and $p = 0.01$.

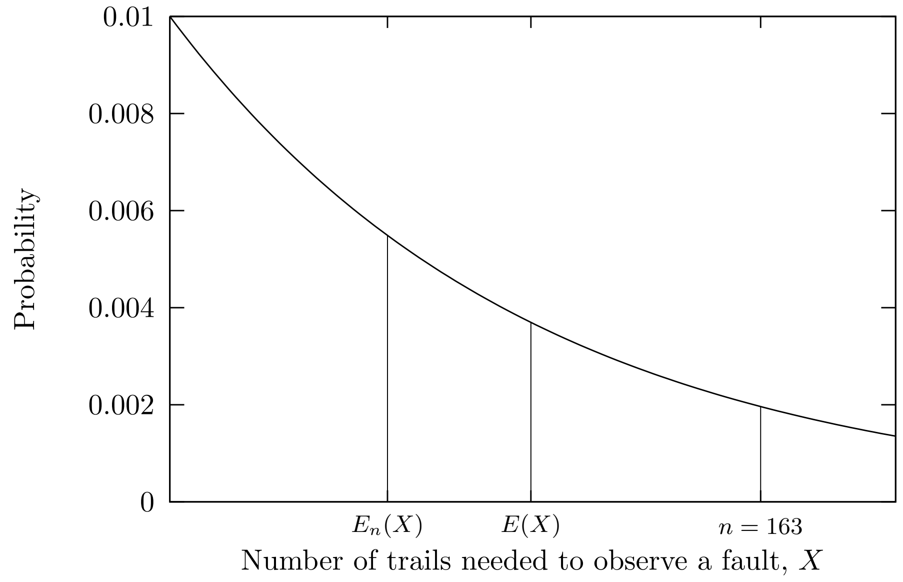

Figure 11: The expected value for the infinite ($E(X) = 100$) vs. the finite case ($E_n(X) = 60.7$)

The wasted iterations would be all the iterations after an error happened, which is $m - E_m(X)$. Their cost is $c(m - E_m(X))$. Hence, the total cost for this case would be $E_{\frac{n}{m}}(Y)c_v + c(m - E_m(X))$. The probability of this case is $\Pr(Y \leq \frac{n}{m}) = 1 - (1-p)^n$ and its total contribution is $\Pr(Y \leq \frac{n}{m})(E_{\frac{n}{m}}(Y)c_v + c(m - E_m(X)))$.

It follows that the expected total cost of error detection with frequent validation can be expressed as follows.

$$\Pr(Y > \frac{n}{m})c_v\frac{n}{m} + \Pr(Y \leq \frac{n}{m})\left(E_{\frac{n}{m}}(Y)c_v + c(m - E_m(X))\right) \tag{1}$$

Given this expression, it is possible to choose the block size m in a way that minimizes the expected cost, as will be illustrated in the examples in Section 3.2.2.

Parallel validation test In the analysis above, it has been assumed that the validation tests are performed sequentially relative to the main computation. However, these tests can be performed partially or completely in parallel, as illustrated in Figure 12. In the case of full parallelism, the time overhead resulting from frequent validation tests would be eliminated at the cost of an increase in the hardware requirements. On the other hand, partial parallelism can be achieved using the idle cycles of the same subunits used for the scalar multiplication.

To model the total cost with full or partial parallel detection, we use a parameter α to measure the amount of overlap between the scalar multiplication and the validation tests. When $\alpha = 0$, there is complete overlap, i.e. full parallelism, while when $\alpha = 1$, the operations are completely serial.

In the case of fully or partially parallel validation, the validation test is performed in parallel with the main computation. If the test determines that the result is valid, the computation is not interrupted. On the other hand, if the test detects an error in the result, the computation of the current block is interrupted. It follows that for all non-faulty blocks, the cost of validation test is αc_v, while for faulty blocks, the cost is still c_v.

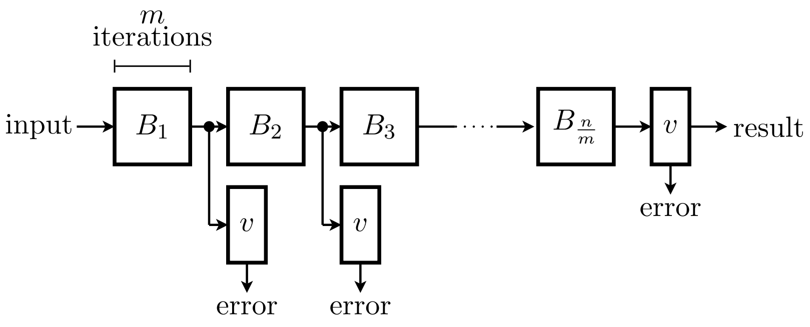

Figure 12: Error detection by frequent validation with parallel testing

Hence, (1) can be modified to model fully and partially parallel tests by modifying the cost of the tests associated with non-faulty blocks, except for the last block.

$$\Pr(Y > \frac{n}{m})\alpha c_v \frac{n}{m} + \Pr(Y \leq \frac{n}{m}) \left(\alpha c_v E_{\frac{n}{m}}(Y) + c(m - E_m(X))\right) + c_v(1-\alpha) \qquad (2)$$

3.2.2 Error Detection with Frequent Validation in ECC

In this section, we discuss the use of frequent validation for error detection in elliptic curve scalar multiplication. For the underlying curve, we will consider elliptic curves on a prime field of size $n = 163$. Also, we will use the conventional double-and-add algorithm as the reference algorithm for scalar multiplication. These ideas are applicable to different curves and scalar multiplication algorithms.

Example 1: Detecting errors caused by invalid-curve and sign-change faults In this example, we analyze the relative advantage of a scalar multiplication unit that performs error detection with frequent validation. We base this design on Algorithm 1, proposed in [3] and discussed in Section 2.2.4. This algorithm can detect errors resulting from both invalid-curve and sign-change faults using a combination of point validation and consistency checking. It also has the added benefit of resisting timing and simple power-analysis attacks.

Algorithm 1 achieves its goal of detecting errors with a simple and efficient test. However, if we consider the fault model presented in Section 3.1, the expected number of iterations needed to observe an error, $E_n(X)$, can be considerably less than n. As an example, when the probability of a fault in an iteration is $p = 0.01$, we get $E_n(X) \approx 60.7$, while for $p = 0.1$, we get $E_n(X) \approx 10$.

This illustrates that if an error can be detected early, time and power can be conserved instead of being used to compute a faulty result. A suitable solution is to check the validity of the intermediate values of Q_0, Q_1 and Q_2 in blocks of size m iterations, where m is chosen to minimize the cost expression given in (1). Algorithm 2 illustrates this approach. Note that in case an error is detected, this algorithm returns the point at infinity as an error signal. Other options are to raise an error flag or an exception condition.

To find an optimal value of m, we note the following, keeping in mind that these values can be tuned for a more accurate model:

- We assume that a point addition and a point doubling have the same cost, which is a common assumption and is particularly true in the case of curves in the Edwards form.

Algorithm 2 Scalar multiplication with frequent validation

```
Input:  $P \in E(K)$, $l = (l_{n-1}, l_{n-2}, l_{n-3}, \ldots, l_0)$, $m \leq n$
Output:  $lP$
 1: $Q_0 \leftarrow \mathcal{O}$, $Q_1 \leftarrow \mathcal{O}$, $Q_2 \leftarrow P$,
 2: for $i = 0$ to $n - 1$ do
 3:     $Q_{l_i} \leftarrow Q_{l_i} + Q_2$
 4:     $Q_2 \leftarrow 2Q_2$
 5:     if $i \bmod m = 0$ or $i = n - 1$ then        ▷ Performing the check for blocks of size $m$
 6:         if $Q_0 \notin E(K)$ or $Q_1 \notin E(K)$ or $Q_2 \neq Q_0 + Q_1 + P$ then
 7:             return $\mathcal{O}$
 8:         end if
 9:     end if
10: end for
11: return $Q_1$
```

Table 1: Cost of frequent validation for Example 1 in point operations

p	Cost at $m = 163$	Optimal m	Cost at optimal m	Expected saving
0.1	310.5	6	15.2	95%
0.01	185.1	21	51.2	72%
0.001	98.4	56	88.8	9.7%

- Each iteration costs 2 point operations.
- The validation test in Algorithm 1 requires checking two points for validity and two point additions. As such, the overall cost of testing is approximately four point operations.

Given these facts and assumptions, we can use (1) to find the optimal block size for different probabilities of fault in iterations. In particular, we have $c_v = 4$, $n = 163$, $c(i) = 2i$. Figure 13 illustrates the cost associated with different values of m at different probabilities of iteration fault, p.

Table 1 shows some examples of the cost saving at different values of p. In particular, the second column states the expected cost, including the validation test and the lost iteration due to errors, associated with a single test at the end of the computation. The third column gives the optimal value of the block size m, while the fourth column gives the expected cost at this value of m. Again, this cost includes all the validation tests and the iteration lost due to errors. The fifth column gives the expected saving due to the optimal choice of m.

We also find the optimal block size and the expected cost for a wide range of iteration fault probabilities, as illustrated in Figure 14. As expected, the optimal block size decreases as the probability of fault increases. Moreover, because of the smaller block sizes, the cost at higher probabilities of fault is not higher than the cost at lower probabilities. In other words, our method succeeds in limiting the loss due to faults in a wide range of probabilities.

To compare with the reference case of a single test at the end of the computation, i.e. $m = 163$, Figure 15 shows the expected and the worst-case cost of both methods. In this case, it is assumed that the worst case is a fault in the first iteration of the last block. It is clear that at a higher

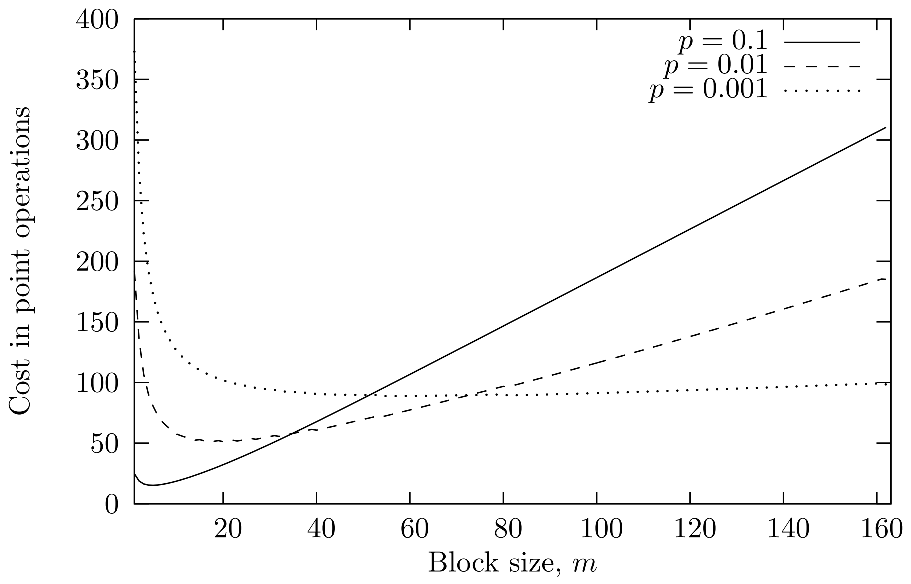

Figure 13: Expected cost of frequent validation in Example 1

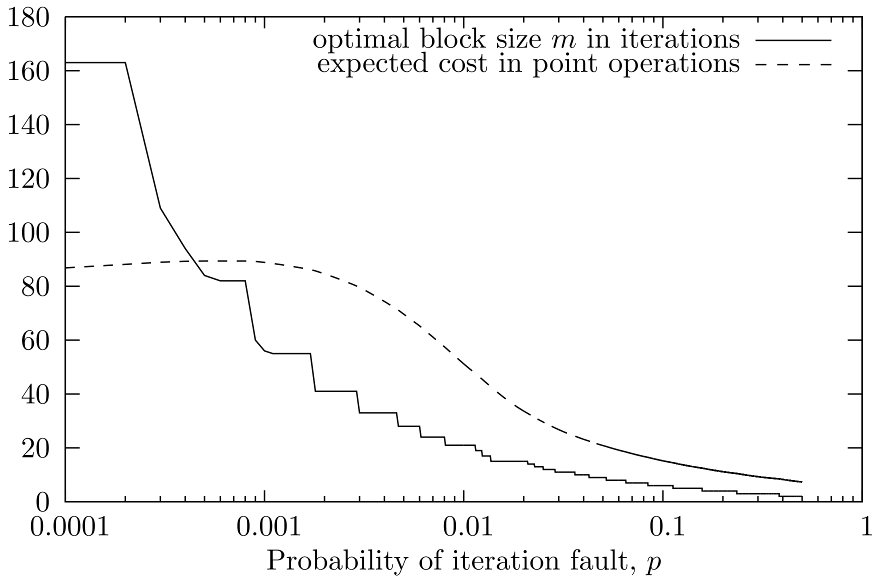

Figure 14: Optimal m and expected cost for different values of p for Example 1

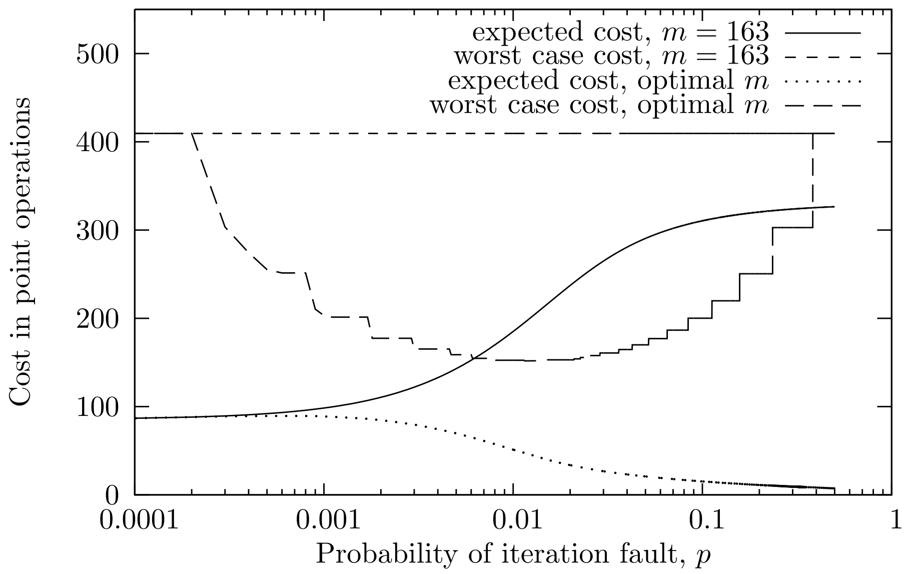

Figure 15: Expected and worst-case cost for optimal m and $m = 163$ in Example 1

probabilities our method shows a significant savings in both the expected and the worst-case cost of detection. This is mainly due to the early detection of faults and the reduction of lost iterations.

Evaluation It can be seen from this example that the use of frequent validation for error detection has a clear advantage over the straight-forward solution. In particular, the early detection reduces the loss due to errors significantly, and in turn, preserves time and reduces power consumption. Moreover, due to the prevention of error propagation, even the worst-case performance is significantly better than the average case of the straight-forward solution for most values of iteration fault probability

Comparing these results with the the structures in Section 2.3.1, we can see that frequent validation reduces the required cost of error detection by reducing the time spent in computation after the occurrence of an error. This effect appears clearly in Figure 15.

4 Frequent Validation with Partial Re-computation for Efficient Error Recovery in ECC

Error recovery is achieved generally by various forms of redundancy. In particular, time redundancy can be effective against transient faults, while hardware redundancy is required to counter permanent faults. Here, we focus on transient faults. In the case of time redundancy, the operation is executed using the same hardware twice or more and the results are checked. When more than two results are available, a majority scheme can be employed to choose the most probable result.

While this approach can be effective against transient faults, its time overhead is high. It can be readily observed that the reason behind this large overhead is the unnecessary repetition of valid computations. We propose the use of frequent validation and partial re-computation as a low-overhead form of time redundancy to achieve efficient fault tolerance in ECSM. In particular,

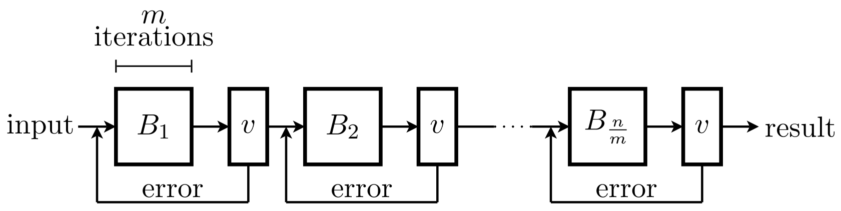

Figure 16: Fault tolerance by frequent validation and partial re-computation

validation tests are employed at specific intervals to detect errors resulting from faults early and recompute only the faulty parts without the need to repeat parts that were computed correctly, as illustrated in Figure 16.

4.1 A Model for Frequent Validation with Partial Re-computation

As mentioned earlier, frequent validation involves dividing iterations into blocks and validating the intermediate results at the end of each block. When an error is detected, only the faulty block is re-computed and the results are tested again. In this section we analyze the cost of the testing and the time overhead of re-computation in this approach. In this context, the overhead includes all operations that are not part of the original scalar multiplication, i.e. validation tests, re-computation and extra point operations. We will also show that the block size can be chosen in a way that minimizes the expected overhead.

As stated in Section 3.1, n denotes the number of iterations, m denotes the block size in iterations, and the probability of an error in an iteration is denoted by p. Moreover, recall that iterations are assumed to be statistically independent. As such, the probability of having m consecutive successful iterations is $\Pr(X \geq m) = (1-p)^m$, and the probability of a faulty block is $\Pr(X < m) = 1-(1-p)^m$. To get the final result, all $\frac{n}{m}$ blocks should be fault-free. Moreover, all faulty blocks have to be repeated, so the total number of blocks executed will depend on the number of faulty blocks detected, but the number of non-faulty blocks will always be $\frac{n}{m}$.

The time overhead in this approach can be divided into two parts: (i) the overhead due to block re-computations, and (ii) the overhead due to testing. We will discuss each one separately.

The time overhead due to block re-computation can be represented as the number of faulty blocks encountered before observing the $(\frac{n}{m})$-th success. Blocks can be represented as Bernoulli trails with probability of success (i.e. probability of one or more errors in the block) as defined above. To model this part of the time overhead, we recall the *Negative Binomial* distribution. A negative binomial distribution with parameters r and p is the probability distribution of the number of failed trails before observing the r-th success in a Bernoulli process with probability of success p for each trail. It has the expected value of $r(1/p-1)$. Thus, the number of faulty blocks before observing the $(\frac{n}{m})$-th success, denoted by Z, will follow a negative binomial distribution with parameters $r=\frac{n}{m}$ and $p=\Pr(X \geq m)$. The expected value of Z is

$$E(Z) = \frac{n}{m}(\frac{1}{\Pr(X \geq m)} - 1)$$

Since each of these blocks has m iterations, it follows that the time overhead due to repeated blocks is $c(n(\frac{1}{\Pr(X \geq m)} - 1))$.

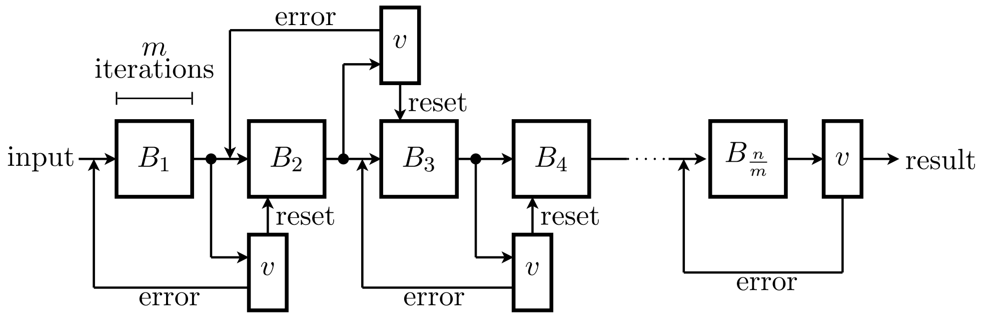

Figure 17: Fault tolerance by frequent validation and partial re-computation with parallel testing

The second part of the time overhead is the overhead due to testing. A test is needed for every block, whether faulty or not. The expected total number of blocks is $\frac{n}{m}\frac{1}{\Pr(X\geq m)}$, so the time overhead due to testing is $c_v\frac{n}{m}\frac{1}{\Pr(X\geq m)}$.

The total overhead can then be expressed as

$$c\left(n\left(\frac{1}{\Pr(X>m)}-1\right)\right)+c_v\frac{n}{m}\frac{1}{\Pr(X>m)} \tag{3}$$

With this expected overhead expression, it is possible to choose the block size m in a way that minimizes the expected overhead of fault tolerance by frequent validation and partial re-computation.

Parallel validation test The validation tests can be performed sequentially, as assumed above. Another option is to perform them partially or fully in parallel, as illustrated in Figure 17, for a significant reduction in the time overhead. In particular, partial parallelism can be achieved without a significant increase in the hardware requirements using the idle cycles of the same subunits used for the scalar multiplication.

Similar to the case of frequent validation, a parameter α is used to model the amount of overlap between the scalar multiplication and the validation tests. When $\alpha = 0$, there is complete overlap, i.e. full parallelism, while when $\alpha = 1$, the operations are completely disjoint.

In the case of full or partial parallel validation, the validation test is performed in parallel with the main computation. If the test determines that the result is valid, the computation is not interrupted. On the other hand, if the test detects an error in the result, the computation of the current block is interrupted and the previous block is re-computed. It follows that for all non-faulty blocks, the cost of validation test is αc_v, while for faulty blocks, the cost is still c_v.

Hence, (3) can be modified to model full and partial parallel tests by modifying the cost of the tests associated with non-faulty blocks, except for the last block.

$$c\left(n\left(\frac{1}{\Pr(X>m)}-1\right)\right)+c_v\left(\frac{n}{m}\left(\frac{1}{\Pr(X>m)}-1\right)+\alpha(\frac{n}{m}-1)+1\right) \tag{4}$$

Reliability As discussed earlier in Section 2.1.3, the reliability of a component is conventionally defined as $R(t) = \Pr(T > t)$, where T is the random variable representing the lifetime of the component and t is the time for which reliability is computed, which is usually taken to be the whole time of the computation. In other words, the reliability is the probability that the component will go through the computation without a fault. This definition of reliability does not apply directly to our approach to fault tolerance, mainly because it was developed to model events of failures and permanent faults [8]. In the case of transient faults, the computation will not necessarily fail when a fault occurs, as it is possible to re-compute the faulty part.

We use a slightly different, but essentially similar, definition of reliability. The reliability of a partial re-computation structure is the probability that the system will produce the correct result within a fixed time threshold. This is essentially similar to the conventional definition since they both measure the probability that the system will perform its function in a given time frame.

In addition to reliability calculations, the time threshold will solve another problem. It is possible that the system will go on indefinitely due to a permanent fault or a determined attacker, so it is important to set a threshold after which the system would always stop and report a failure.

This threshold can be determined using the cumulative distribution function (CDF) of Z, the number of faulty blocks before observing the $(\frac{n}{m})$-th success. A certain, usually high, value of the CDF is chosen and the corresponding overhead is set as a threshold for the computation. This threshold gives a trade-off between the component reliability and the tolerance overhead. For example, for a reliability of 99%, we find the value z of Z for which $\mathrm{CDF}(z) = 0.99$. Then, the threshold is set to the overhead associated with z. However, this only applies to transient faults since no amount of re-computation can recover from permanent faults. As such, the reliability of our method against permanent faults is the same as a stand-alone scalar multiplication unit. The $\mathrm{CDF}(k)$ of a random variable that has a negative binomial distribution with parameter r and a trail success probability p is computed as $I_p(r, k+1)$, where $I_p(x, y)$ is the regularized incomplete beta function computed as follows

$$I_p(x,y) = \sum_{j=x}^{x+y-1} \frac{(x+y-1)!}{j!(x+y-1-j)!} p^j (1-p)^{x+y-1-j} \tag{5}$$

4.2 Frequent Validation with Partial Re-computation in ECSM

In this approach, fault tolerance is achieved by employing an efficient error detection scheme and performing frequent validation tests. When a fault is detected in a block, the block is re-computed and checked again. For a block size equal to n, this means repeating the whole scalar multiplication. However, if smaller blocks are used, less re-computation is required.

Example 2: Fault tolerance with frequent validation and partial re-computation in ECSM This example is analogous to Example 1 in Section 3.2.2. In this example, both types of faults, namely, invalid-curve and sign-change faults, will be considered. As countermeasures, point validation and consistency checking will be used as described Algorithm 3.

Following the same cost assumptions stated in Example 1 in Section 3.2.2, we can use the expression in (3) to find an optimal value of the block size m that minimizes the overhead. Figure 18 illustrates the overhead associated with different block sizes and probabilities of iteration faults.

Table 2 summarizes the result for different values of p. The second column gives the component reliability associated with the probability of iteration fault, p, while the fourth and fifth columns give the overhead in terms of point operations and relative to a scalar multiplication with no error

Algorithm 3 Scalar multiplication with frequent validation and partial re-computation

```
Input:  $P \in E(K)$, $l = (l_{n-1}, l_{n-2}, l_{n-3}, \dots, l_0)$, , block size $m$
Output:  $lP$
 1: $Q_0 \leftarrow \mathcal{O}$, $Q_1 \leftarrow \mathcal{O}$, $Q_2 \leftarrow P$
 2: $j \leftarrow 0$, $H_0 \leftarrow Q_0$, $H_1 \leftarrow Q_1$, $H_2 \leftarrow Q_2$
 3: for $i = 0$ to $n - 1$ do
 4:     $Q_{l_i} \leftarrow Q_{l_i} + Q_2$
 5:     $Q_2 \leftarrow 2Q_2$
 6:     if $i \bmod m = 0$ or $i = n - 1$ then          ▷ perform the check for blocks of size $m$
 7:         if $Q_0 \in E(K)$ and $Q_1 \in E(K)$ and $Q_2 = Q_0 + Q_1 + P$ then
 8:             $j \leftarrow i$, $H_0 \leftarrow Q_0$, $H_1 \leftarrow Q_1$, $H_2 \leftarrow Q_2$          ▷ store the current state
 9:         else
10:             $i \leftarrow j$, $Q_0 \leftarrow H_0$, $Q_1 \leftarrow H_1$, $Q_2 \leftarrow H_2$          ▷ restore the previous correct state
11:         end if
12:     end if
13: end for
14: return $Q_1$
```

Table 2: Overhead of frequent validation and partial re-computation in point operations for Example 2

p	Component reliability	Optimal m	Overhead at optimal m		Reliability, optimal m
			point operations	relative to SM	
0.0001	98.3%	140	90.8	37.1%	~100%
0.001	84.9%	44	111.7	45.6%	99.9%
0.01	19.6%	13	184.2	75.3%	76.1%

detection or fault tolerance, respectively. This means that all the tests and re-computations are included in the overhead. The last column gives the reliability of the design using optimal m, which corresponds to the least expected overhead.

It can be observed that, as expected, the optimal block is large for very low values of p while for high probabilities of fault, smaller block sizes give much smaller overhead. Figure 19 shows the optimal block size, m, and the expected overhead associated with different values of p.

The cumulative distribution function can be used to measure the reliability of the structure. Figure 20 shows the reliability of optimal block size at different values of p. It can be seen that the reliability never falls below 0.5, which is a property of the negative binomial distribution. When higher reliability is required, a threshold that is higher than the expected value can be set such that the CDF of that threshold is greater than or equal to the required reliability. A threshold also helps in preventing infinite loops.

Figure 21 shows the threshold associated with a reliability of 0.9 for different values of p. It illustrates that a reliability of 90% can be expected using less than 200% overhead for values $p < 0.05$, which is a very high value of iteration fault probability as illustrated in Figure 9. This is an indication of the advantage of partial re-computation in maintaining high reliability in an unfavorable situation with a relatively small overhead.

An interesting observation from Figure 21 is the behavior of the gap between the expected

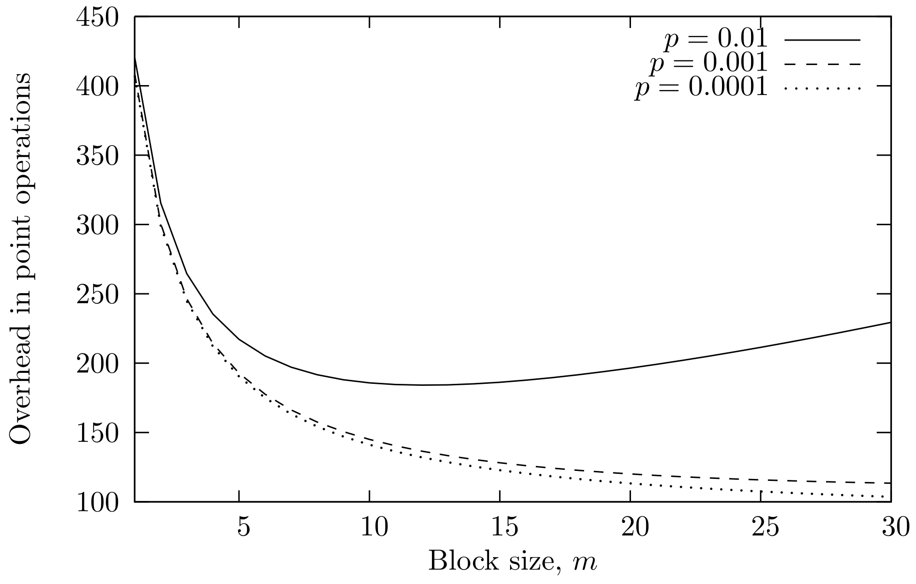

Figure 18: Expected overhead of frequent validation and partial re-computation in Example 2

overhead at optimal m and the reliability threshold. This gap starts considerably large at low values of p and shrinks as the value of p increases. The main reason for this behavior is that the choice of the block size m takes into consideration only the expected overhead. It follows that, for lower values of p, the chosen value of m is quite large, as illustrated in Figure 19. Large values of m significantly increase the reliability threshold since a block re-computation becomes expensive.

So, in order to reduce the expected cost of a highly-reliable structure, we propose to optimize the value of m according to the reliability threshold rather than the expected overhead. This will make a significant difference only at lower values of p, as illustrated in Figure 22, which shows both the overhead-optimized and the reliability-optimized values of m at different values of p.

It is clear that the expected overhead of a reliability-optimized m would be higher than the overhead-optimized m. However, the gain appears when considering the reliability thresholds of both as illustrated in Figure 23.

4.3 Evaluation and Comparison

In this section, we evaluate the results of the preceding example and compare its results to the structures outlined in Section 2.3.2.

Fault tolerance by partial re-computation can be classified as a limited time redundancy solution, where only faulty blocks are recomputed. The effect of the limited redundancy is clear in the low expected cost reported in Example 2, even for relatively high probabilities of iteration fault. An interesting observation is that the reliability of the structure does not drop below 0.5 as the iteration fault probability increases, but the overhead grows much faster with higher values of probability of iteration fault. However, we recall that the practical region of iteration fault probability is $p < 0.05$ as illustrated in Figure 9.

The allowed redundancy can be adjusted to satisfy certain reliability constraints at relatively high probabilities of iteration fault. As illustrated in Figure 21, which shows the threshold associ-

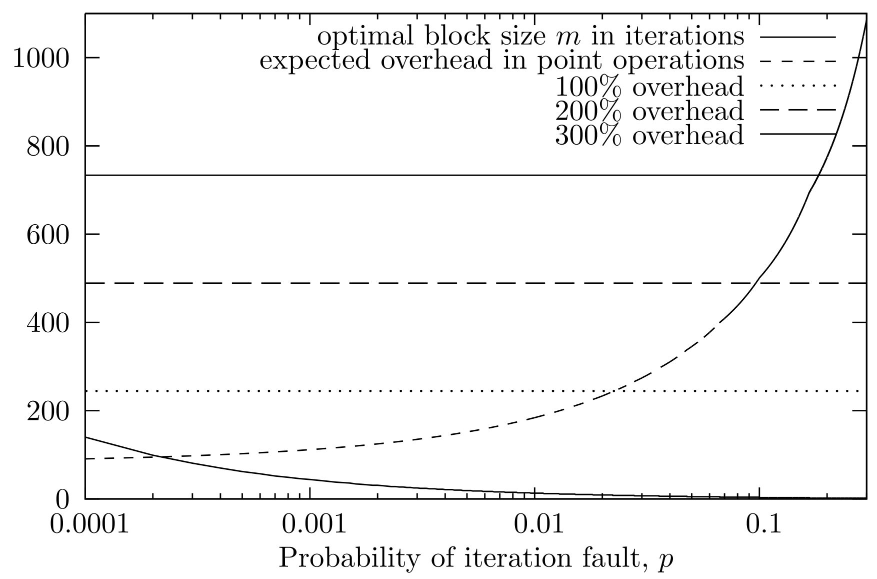

Figure 19: Optimal m and expected overhead for different values of p in Example 2

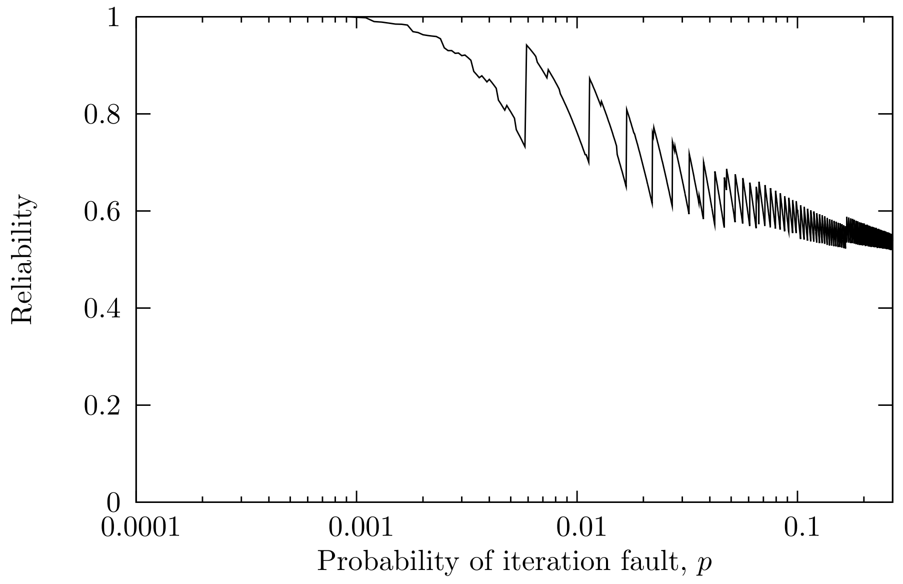

Figure 20: Reliability for optimal m for different values of p in Example 2

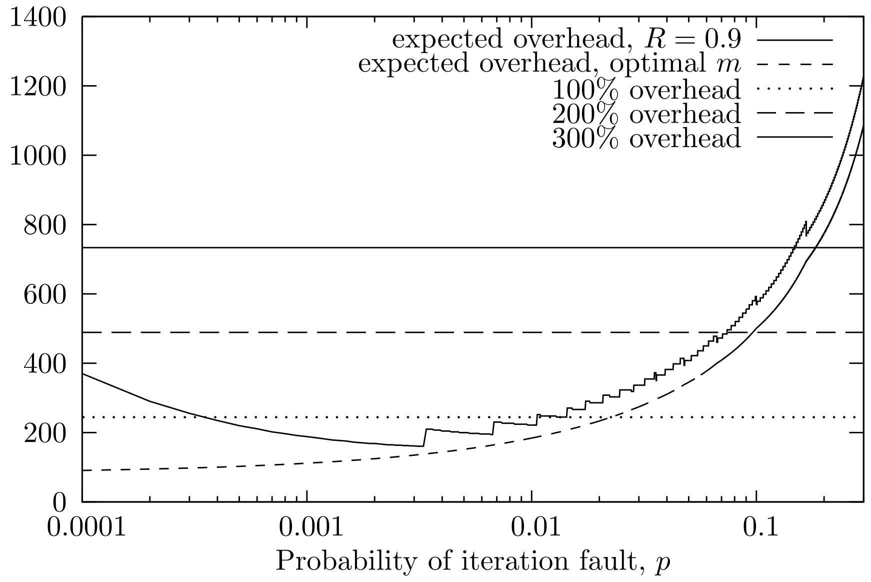

Figure 21: Overhead threshold associated with $R = 0.9$ for different values of p in Example 2

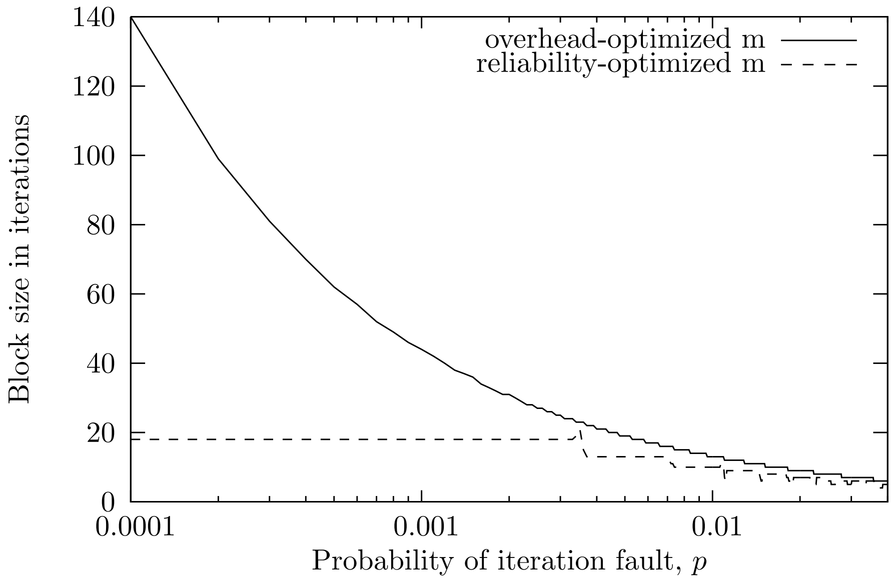

Figure 22: Overhead-optimized and reliability-optimized values of m in Example 2

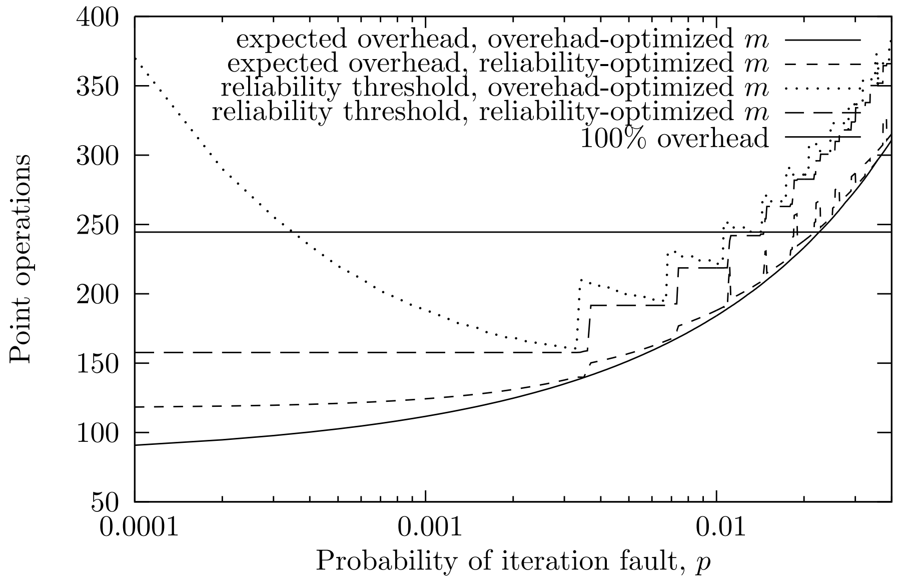

Figure 23: Expected overhead and reliability threshold for overhead-optimized and reliability-optimized values of m in Example 2

ated with a reliability value of 90%, the required threshold is relatively low even at high probabilities of iteration fault. This is due to frequent validation and the small block size. Also, at low probabilities of iteration faults, where large block sizes significantly increase the reliability threshold, the block size can be chosen in a way that reduces the reliability threshold with a limited effect on expected overhead.

We will compare known solutions to examine the effects of frequent validation with partial re-computation on the performance and reliability of fault tolerance structures. The solutions considered include triple-modular redundancy (TMR), double-modular redundancy with point validation (DMR-PV) and parallel computation with re-computation (PRC), all of which have been reviewed in Section 2.3.2. Also, we consider the structure in Example 2 in two variates, overhead-optimized and reliability-optimized. All of these solutions are evaluated relative to a stand-alone scalar multiplication with no error detection or fault tolerance capabilities.

Reliability We begin by comparing the structure reliability of each of the methods relative to the reliability of the scalar multiplication unit. As Figure 24 illustrates, the reliability of conventional redundancy schemes falls rapidly with the reliability of the scalar multiplication unit. However, the reliability of the frequent validation with partial re-computation scheme does not fall as quickly. This is caused mainly by the small block sizes at higher probabilities of fault and by the limited re-computation of only the faulty blocks. Note that this reliability estimate is limited to transient faults since re-computation can not help recover from permanent faults.

Time overhead We also compare the expected time overhead of various solutions. TMR and DMR-PV are not included in this comparison because they are hardware redundancy schemes and their time overhead is almost negligible. As Figure 25 illustrates, the expected time overhead of all schemes grows with the probability of fault. However, when we examine the practical range

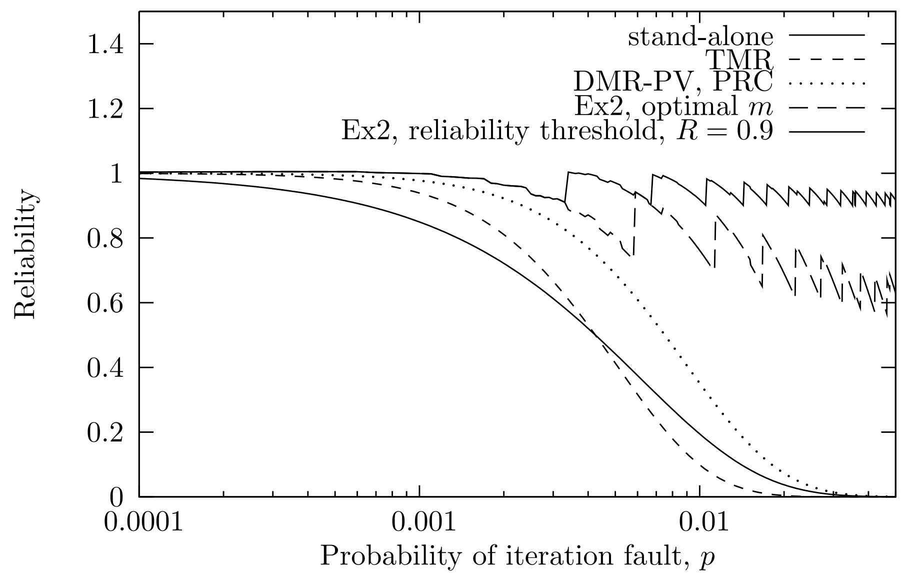

Figure 24: Reliability of various schemes at different values of p

of iteration fault probabilities, we can see that the expected time overhead of frequent validation with partial re-computation scheme is better than other schemes based on full time redundancy. Moreover, while the reliability of PRC falls rapidly with the increase in p, the structure in Example 2 maintains a reliability 90% with the reliability threshold.

Hardware overhead Schemes that depend on hardware redundancy, like TMR, DMR-PV and PRC, have the advantage of tolerance to permanent faults. However, this comes at the cost of doubling the hardware requirements in the case of DMR-PV and PRC, and tripling it for TMR.

On the other hand, schemes based on frequent validation with partial re-computation have no significant hardware overhead, and are only able to detect, but not tolerate, permanent faults. For this reason, these schemes are optimal for environments where space is scarce and high reliability is required.

4.4 Effects of Frequent Validation on Security

The use of frequent validation in error detection and fault tolerance may introduce variability in the timing of the operations, and a potential attacker may be able to gain some information. In this section we discuss these effects on the security of the considered operation.

If the time required to execute a cryptographic operation changes depending on the secret information used in the computation, this variability can be exploited to discover the secret information totally or partially. As such, it is essential to make the time taken by a cryptographic computation independent from any secret information. Many methods to achieve this have been already proposed.

This approach to error detection and fault tolerance can be applied to both equal-time and variable-time iterative computations. It introduces its own variability on top of the underlying algorithm, since the time taken to abort the computation (in the case of error detection) or finish

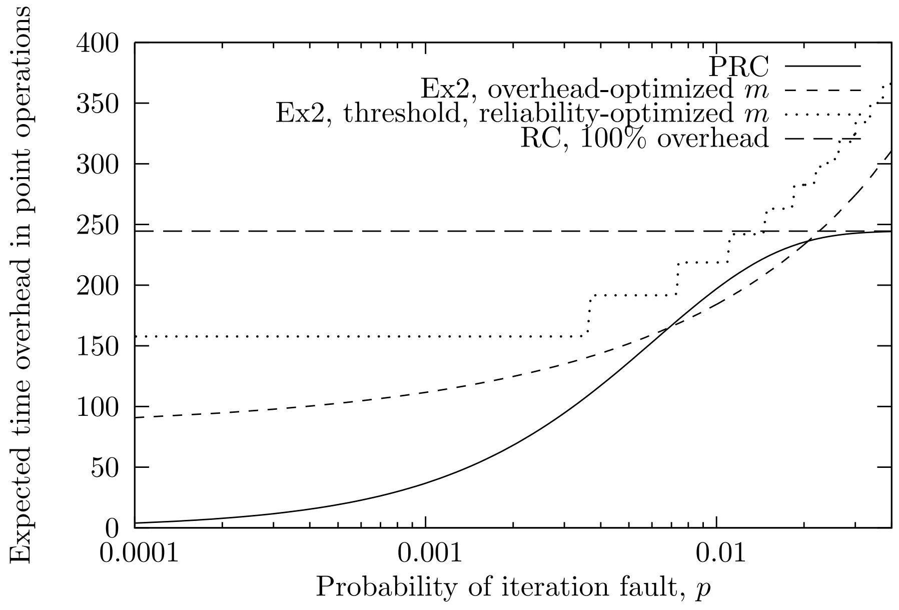

Figure 25: Expected time overhead of various schemes in point operations at different values of p

it (in the cast of fault tolerance) is variable.

In the case of fixed-time algorithms, the variability introduced by this approach will not undermine the security of the algorithm. This is due to the fact that all the variability introduced by this approach is based on the frequency and positions of faults, and is independent from the secret information.

In the case of variable-time algorithms, the underlying algorithm is already exposed to timing attacks, and for the same argument above, this approach will not make it any easier for the attacker to discover the secret information.

5 Conclusion

In this work, we have proposed the use of frequent validation in error detection and recover for transient faults for elliptic curve scalar multiplication systems. Most previous proposals dealt with the scalar multiplication as a black box and considered only testing the inputs/outputs for error detection and time or space redundancy for fault tolerance.

In our approach, we divide the scalar multiplication iterations into blocks and we use simple and efficient error detection schemes to detect errors early and reduce the loss due to faults. Moreover, we use the same error detection schemes with partial re-computation to achieve efficient error recovery without requiring complete time or hardware redundancy. The analysis and examples given illustrate that the use of frequent validation is considerably more efficient and reliable than known error detection and fault tolerance schemes especially when faults are frequent or in the case of fault attacks. Thus, for certain scenarios, the use of frequent validation in error detection and fault tolerance may be a promising approach.

Some of the possible extensions of the work presented here include the use of input encoding and randomization in the re-computation of faulty blocks and the effect of existing hardware redundancy. Another issue that might be of importance is the off-line choice of the block size which involves making assumptions about statistical properties of faults.

References

[1] Adi Shamir. Method and apparatus for protecting public key schemes from timing and fault attacks. US Patent 5,991,415, November 1999.

[2] Agustin Domínguez-Oviedo and M. Anwar Hasan. Improved error-detection and fault-tolerance in ECSM using input randomization. Technical report, CACR 2006-41, University of Waterloo, 2006. A revised version to appear in IEEE Transactions on Dependable and Secure Computing.

[3] Agustin Domínguez-Oviedo and M. Anwar Hasan. Algorithm-level error detection for ECSM. Technical report, CACR 2009-05, University of Waterloo, 2009.

[4] Kyle Siegrist *et al.* Virtual laboratories in probability and statistics: Exponential distribution. [Online; accessed 10-April-2008].

[5] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. Technical Report 2004/100, Cryptology ePrint Archive, 2004.

[6] Ingrid Biehl, Bernard Meyer, and Volker Muller. Differential fault attacks on elliptic curve cryptosystems. In *CRYPTO'00: Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, pages 131–146. Springer-Verlag, 2000.

[7] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. Sign change fault attacks on elliptic curve cryptosystems. Technical Report 2004/227, Cryptology ePrint Archive, 2004.

[8] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems.* Morgan Kaufmann, 2007.

[9] Mathieu Ciet and Mark Joye. Elliptic curve cryptosystems in the presence of permenant and transient faults. Technical Report 2003/028, Cryptology ePrint Archive, 2003.

[10] U.S. Department of Defense. *Military Standerization Handbook: Reliability Prediction of Electronic Equipment, MIL-HDBK-217F(2)*, February 1995.