Adaptive Error Recovery for Transient Faults in Elliptic Curve Scalar Multiplication

Abdulaziz Alkhoraidly and M. Anwar Hasan
University of Waterloo

Abstract

The use of fixed-block error recovery, which combines frequent validation and partial recomputation, to address the problem of transient faults in elliptic curve scalar multiplication was proposed earlier and its advantages in terms of efficiency and reliability were illustrated. However, in order to maximize its advantages, the selection of the block size has to be optimized, which requires knowledge of the statistical properties of errors. It was shown that this can be partially alleviated by selecting smaller block sizes. We introduce an alternative approach that aims to reduce the dependency on prior knowledge. Instead of using a fixed block size, we propose the use of an adaptive block size that varies depending on whether or not an error is detected. The performance of this approach is studied using an analytical model and simulation under constant and variable error rates and the results show that it can approach, and in some cases exceed, the performance of the fixed-block error recovery approach while not requiring prior knowledge of the statistical properties of errors.

1 Introduction

Elliptic Curve Cryptography (ECC) was pioneered independently by Miller [12] and Koblitz [10], and since then it has gained wide acceptance in both standardization and practice. The main advantage of an elliptic curve cryptosystem relative to the earlier-invented RSA cryptosystem [14] is that the best known general mathematical attacks on ECC have exponential complexity. This leads to better efficiency since smaller system parameters to be used to achieve the same level of security [6].

The main operation in an ECC-based system is the Elliptic Curve Scalar Multiplication (ECSM), which dominates the computation's time. During this operation, natural or deliberate faults can occur in the underlying hardware, which can be either a general-purpose processor or a dedicated accelerator. The resulting faulty results can then be used to learn the secret information partially or fully. We focus our attention on *random, transient faults* which occur due to natural causes or malicious acts of a *less-sophisticated* attacker, i.e., one who has a limited control over the timing and location of the injected faults. A variety of techniques have been proposed to deal with these transient faults. Some of these techniques work by preventing the occurrence of faults while others attempt to detect resulting errors before the faulty results are presented as output and withhold or mask the output. Moreover, these techniques can be used along with different types of redundancy in order to allow for error recovery and achieve higher reliability.

When a fault occurs, all computations performed between its occurrence and its detection are corrupted and their results will have to be discarded. This is a significant loss in time and power especially for an operation like ECSM, which has hundreds of iterations each of which involves a relatively large number of finite field operations on operands that are hundreds of bits in width. As an example, a scalar multiplication using projective coordinates on an elliptic curve defined over a 256-bit prime field requires about 4600 field operations (multiplication and squaring) of 256-bit elements [6]. If a fault occurs near the middle of that scalar multiplication and no validation tests are performed until the end, the processing time and power corresponding to about 2300 field multiplication and squaring operations will be lost. Any reduction in this loss will have a positive impact on the overall efficiency of the system, especially for light-weight and mobile platforms. For comparison, an exhaustive error detection test costs about 56 field operations [7].

This issue has been addressed in [1] where the use of frequent validation for error detection and recovery was proposed. In essence, the scalar multiplication operation is partitioned into equal blocks of iterations and the output of each block is validated before starting the next one. If an error is detected in the output of a block, that block is recomputed. It has been shown that error recovery based on frequent validation is more reliable and has less overhead compared to conventional error detection and recovery designs especially when errors are frequent.

However, a practical challenge of the approach proposed in [1] is that the selection of the overhead-optimal (or reliability-optimal) block size depends on the statistical error model. While this is acceptable in a theoretical setting, a more practical approach is to reduce or eliminate the need for an accurate model for errors. In this work, we propose an extension to

[1] by which a statistical model for errors is no longer needed to achieve efficient error recovery. Instead of fixing the block size to an optimal value set in advance, we allow the block size to vary adaptively in a given range based on whether an error is detected or not. We show that adaptive error recovery can approach the efficiency and reliability of optimized fixed-block error recovery.

This document is organized as follows. In Section 2, we give a brief overview of elliptic curves and their use in cryptography. We also review known fault analysis attacks and discuss error handling in elliptic curve implementations. Section 3 discusses adaptive error recovery and describes an analytical model that can be used to estimate the expected overhead of this approach under a constant error rate. These estimates are confirmed with simulation. In Section 4, simulation is used to explore the performance of adaptive error recovery under a more general error model with variable error rate. Section 5 discusses the effects of adaptive error recovery on the security of the ECSM design, and how some of the known security issues can be addressed.

2 Background

In this section, a brief overview of elliptic curves and their use in cryptography is given. An extended treatment of this subject can be sought in references like [6]. Also, causes of faults and known methods to exploit resulting errors, along with techniques to handle said errors, are discussed.

2.1 Elliptic Curve Cryptography

An elliptic curve E defined over a field F is the set of points $(x, y) \in F^2$ that satisfy the Weierstrass equation,

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{1}$$

where all coefficients are elements in F and such that the curve is nonsingular, i.e., the partial derivatives do not vanish simultaneously at any point. In the case of prime finite fields, (1) can be simplified to

$$E : y^2 = x^3 + ax + b \tag{2}$$

The points on an elliptic curve, together with the point at infinity $\mathcal{O}$, form an abelian group under the operation of *point addition*. The operation of computing the l^{th}-multiple of a point $P \in E$, i.e., adding P to itself l times,

Algorithm 1 Binary double-and-add scalar multiplication

```
Input: $P \in E(F)$, $l = (l_{n-1}, l_{n-2}, l_{n-3}, \dots, l_0)$
Output: $lP$
1: $Q = \mathcal{O}$
2: for $i = n - 1$ downto $0$ do
3:     $Q = 2Q$
4:     if $l_i = 1$ then
5:         $Q = Q + P$
6:     end if
7: end for
8: return $Q$
```

denoted by lP, is called a *scalar multiplication.* Scalar multiplication is the key primitive in ECC, and is analogous to the modular exponentiation operation employed in RSA. Its execution time usually dominates the time required to perform ECC operations.

The security of ECC is based on the hardness of the Elliptic Curve Discrete Logarithm Problem (ECDLP), which can be defined as finding the scalar l given the points $P, lP \in E$. Since the best known solutions for this problem in general elliptic curves have an exponential time complexity, the ECDLP seems to be significantly more difficult than the problems of integer factorization and the DLP in a finite field. As such, a comparable security level can be achieved using significantly smaller system parameters, which gives ECC an advantage in terms of efficiency in both software and hardware implementations.

Scalar multiplication can be naively performed by repeated addition. This, like the repeated multiplication in the case of exponentiation, requires a time that is exponential in the number of bits in the representation of the scalar. A better solution, which has a linear time complexity, is to use the double-and-add approach as shown in Algorithm 1.

2.2 Faults in Elliptic Curve Cryptosystems

Faults can occur in a device either naturally or as a result of a deliberate action, and can be caused by one of many reasons. For instance, they can be caused by variations in standard operation conditions like supply voltage, clock frequency or operating temperature [3]. Faulty results can then be used to learn the secret information partially or fully through Fault Analysis Attacks (FAAs), which can be generally classified into the following classes:

1. Invalid-curve Attacks: In this class of fault attacks, the attacker attempts to move the computation from the strong curve to a different, probably weaker, curve by injecting faults either in the base point, the curve parameters or during the computation. The resulting incorrect output values can be used to guess intermediate values in the computation, which can reveal parts of the secret key. Usually, the attack has to be repeated since in many cases the value guessed is not unique. Among the attacks in this class are the variations presented by Biehl et al. in [4], Antipa et al. in [2] and Ciet et al. in [5].

2. Sign Change Attack: While earlier fault attacks on ECC worked by inducing faults in a way that would move the computation to a different elliptic curve, the attack described by Blomer et al. in [9] results in a faulty point that still belongs to the original curve. As the name indicates, this attack works by targeting the sign of intermediate points during the computation. By collecting enough of these faulty results, the secret key can be recovered in expected polynomial time.

3. Safe-error Attacks: These attacks target scalar multiplication algorithm where dummy intermediate variables are used to thwart timing and simple power analysis attacks, and where the output values are checked for errors [15]. The principle is to inject a fault during an iteration of the algorithm and observe whether the resulting error is detected or not. This will reveal whether the computation performed in that iteration was a dummy computation, and consequently expose the corresponding bit of the key.

2.3 Error Handling Techniques

There is a variety of known techniques that can be used to handle data errors caused by faults. Generally, these techniques work by either preventing the injection of faults, detecting the resulting errors, or masking the faulty result randomly. It is also possible to extend detection techniques by time or hardware redundancy to allow for error recovery. While some of these techniques are only applicable to ECC or to specific classes of faults, others apply more generally.

2.3.1 Error Prevention, Detection and Masking

The occurrence of some types of faults can be prevented through physical means like *sensors* and *metal shields* [3]. Moreover, sign change fault attacks

Algorithm 2 Double-and-add-always SM algorithm with point validation and consistency checking

```
Input:  $P \in E(F)$, $l = (l_{n-1}, l_{n-2}, l_{n-3}, \dots, l_0)$
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 $Q_0 \in E(F)$ and $Q_1 \in E(F)$ and $Q_2 = Q_0 + Q_1 + P$ then
 7:     return $Q_1$
 8: else
 9:     return $\mathcal{O}$
10: end if
```

can be prevented using *Montgomery's scalar multiplication algorithm* [13] since it does not use the y-coordinate, and hence does not allow sign change.

As for detection, *point validation* can be used to detect invalid-curve errors as the representation of an elliptic curve point has some inherent information redundancy [4]. It is also possible to use *time* and/or *hardware redundancy*, accompanied by comparison, to detect faulty results [8]. *Randomization* can also be used in ECSM in a variety of ways while encoding the scalar, base point or curve parameters. Combined with hardware or time redundancy, it prevents similar errors from generating similar faulty results thus aiding in the detection of errors [8]. Most notably, some algorithms involve redundant computations that allow for detecting errors by *checking the consistency* of the results. For example, in Algorithm 2, intermediate variables satisfy a set of invariants that can be used to check for consistency. These invariants allow for detection of a wide range of errors including those resulting from the three classes of FAAs discussed earlier [7].

It is also possible to use some techniques to mask the faulty results. For example, *randomization* is effective in masking some types of errors, particularly those resulting from sign change faults [9]. Moreover, since a validation test is a logical test, its outcome can be manipulated by flipping a single bit, effectively becoming a single point of failure. *Infective computation*, as presented in [16], avoids this by replacing a validation test with a computation that allows the correct result to pass unchanged and masks the faulty result randomly. This way, the attacker can not use the faulty result since it is no longer correlated with the secret information.

2.3.2 Error Recovery

It is also possible to combine error detection with time or hardware redundancy, as mentioned earlier, to allow for error recovery. One such approach is the use of *N-modular redundancy*, e.g., Triple Modular Redundancy (TMR). Triple modular redundancy with a majority vote works well when faults are limited to one block. When two or more blocks produce different faulty results, it can detect faults but cannot determine the correct result. However, an attacker can inject the same fault in two or more modules and make the structure output a faulty result. This issue has been addressed in [8] using randomized input encoding, which results in randomized computations and similar faults causing different faulty results.

Another method for combining hardware redundancy with information redundancy is Dual Modular Redundancy with Point Validation (DMR-PV) [8]. This is a simple replication with comparison scheme combined with point validation. The inputs to each of the modules are randomly encoded and the results of each are tested by a point validation module. This structure can recover from both natural errors and errors caused by an invalid-curve fault attack limited to one module, and can 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 bypass the validation tests by injecting a sign change fault in one of the modules and a random fault in the other.

It is also possible to combine time and hard ware redundancy as in Parallel Computation with Recomputation (PRC) [8]. This scheme is a combination of multiple types of redundancy, namely hardware redundancy, time redundancy and randomized input encoding. Two modules are used and their inputs are encoded, then their results are compared. If the results are not equal, 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.

2.4 Fixed-block Error Recovery

While the error recovery designs discussed earlier have their merits, they generally have two disadvantages:

1. Errors are allowed to propagate until the end of the computation where data validity is tested. Since all computations performed after the

occurrence of an error are wasteful, it can be more efficient to limit this loss.

2. When recomputation is required, full recomputation causes unnecessary repetition of error-free iterations. It can be more efficient to limit recomputation to faulty iterations.

Both issues can be addressed using frequent validation with partial recomputation. In particular, fixed-block error recovery [1] can be classified as a partial time-redundancy approach and works specifically for errors resulting from transient faults. Instead of full time-redundancy, the computation is divided into equal blocks and a validation test is performed for the intermediate variables at the end of each block. Then, only faulty blocks are recomputed as illustrated in Algorithm 3. This limits both unnecessary recomputations and the loss due to error propagation to subsequent iterations, and enhances the overall reliability of the design.

Different criteria can be used to select the block size. For instance, when it is assumed that errors can be modeled statistically, the block size can be selected to minimize the expected overhead, which includes the costs of both testing and recomputation. Moreover, the block size can be chosen to minimize the overhead associated with a given reliability requirement, which has been shown to lead to results that are less sensitive to the statistical error model. In both cases, it has been illustrated in [1] that fixed-block error recovery has less overhead and higher reliability compared to earlier approaches.

Despite its advantages, fixed-block error recovery requires some knowledge of the statistical properties of error to achieve optimal performance. This is because the choice of the optimal block size depends on the knowledge of the error rate, which is not always available in practice. This can be mitigated by overestimating the error rate which leads to a smaller block size and a generally more reliable system. However, this will increase the cost of testing unnecessarily especially when errors are infrequent.

3 Adaptive Error Recovery under a Constant Error Rate

In this section, we describe adaptive error recovery and present an analytical model that can be used to estimate its expected overhead. Then, a numerical example is presented and used, along with simulation, to validate the analytical results.

Algorithm 3 Scalar multiplication with frequent validation and partial recomputation

```
Input: $P \in E$, $l = (l_{n-1}, l_{n-2}, l_{n-3}, \dots, l_0)$, 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 $Q_0 \in E$ and $Q_1 \in E$ 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 correct state
11:         end if
12:     end if
13: end for
14: return $Q_1$
```

3.1 Adaptive Error Recovery

While fixed-block error recovery has some advantages over comparable approaches to fault tolerance, it has the disadvantage of requiring the optimal (or reliability-optimal) block size to be determined prior to the computation based on the knowledge of the error rate. As an alternative, we propose adaptive error recovery, which employs a similar idea but where the block size is varied in response to the perceived error rate, resulting in less validation tests when errors are rare and more tests when errors are more frequent. While this approach can be less efficient than the fixed-block approach for a specific, known error rate, it will require no prior knowledge of the error rate and probably perform better in error rates different from the one for which an optimal fixed block size was selected.

Specifically, adaptive error recovery differs from fixed-block error recovery in the following aspects:

1. Rather than being constant, the block size, denoted by m, varies within a given range, $[m_l, m_h]$.
2. The variation of m is controlled by an adaptive policy based on whether or not an error is detected.

As a result, the number of iterations completed successfully does not progress in regular increments like before, but changes by an amount and probability that both depend on the current m. Algorithm 4 illustrates the adaptive error recovery approach. Note the use of two functions, *mUp* and *mDown*, that define the adaptive policy based on the occurrence or absence of errors. Although these functions can be chosen arbitrarily, it is generally more efficient to limit them to functions that are easily provided by a hardware register, e.g., an increment or a shift, rather than functions that require additional time and/or space.

3.2 Modeling Adaptive Error Recovery

In this section, we describe a model to calculate the time overhead associated with adaptive error recovery. This model is similar in essence to the one presented in [1], but is superior in that it can be used to estimate the overhead of both fixed-block and adaptive error recovery.

Algorithm 4 Scalar multiplication with adaptive error recovery

Input: $P \in E$, $l = (l_{n-1}, l_{n-2}, l_{n-3}, \dots, l_0)$, $[m_l, m_h]$ block size range, m_{init} block size initial value

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: $m_c \leftarrow m_{\text{init}}$
4: for $i = 0$ to $n-1$ do
5:     $Q_{l_i} \leftarrow Q_{l_i} + Q_2$
6:     $Q_2 \leftarrow 2Q_2$
7:     if $i - j = m_c$ or $i = n-1$ then                          ▷ test at end of block
8:         if $Q_0 \in E$ and $Q_1 \in E$ and $Q_2 = Q_0 + Q_1 + P$ then ▷ error-free
9:             $j \leftarrow i$, $H_0 \leftarrow Q_0$, $H_1 \leftarrow Q_1$, $H_2 \leftarrow Q_2$
10:            $m_c \leftarrow mUp(m_c, m_h)$      ▷ increase block size with upper limit $m_h$
11:        else                                                    ▷ error detected
12:            $i \leftarrow j$, $Q_0 \leftarrow H_0$, $Q_1 \leftarrow H_1$, $Q_2 \leftarrow H_2$
13:            $m_c \leftarrow mDown(m_c, m_l)$ ▷ decrease block size with lower limit $m_l$
14:        end if
15:    end if
16: end for
17: return $Q_1$
```

3.2.1 Assumptions and Notation

In this work, we deal with errors caused by *transient* faults that affect an ECSM implementation. In particular, we use the fact that the state of the algorithm, i.e., intermediate results, can be tested for validity after each iteration and that the validation tests are relatively efficient. When an error occurs anywhere in an iteration of the algorithm, it corrupts the whole iteration. Moreover, an error will propagate through consecutive iterations and the final result will be invalid.

In what follows, we describe the statistical error model used in this work. A more complete description can be found in a reference on reliability and fault tolerance like [11]. Let n denote the number of iterations, and let X be a random variable that describes the time (in iterations) until the next error occurrence. The rate of errors per unit time, where time is measured in iterations, is denoted by λ. When λ is constant, X is known to follow an exponential distribution with rate parameter λ. Also, it follows that the probability of an error occurring on or before time x is $\Pr[X \leq x] = 1 - e^{-\lambda x}$, while the reliability for time x is defined as $\mathrm{Rel}(x) = \Pr[X \geq x] = e^{-\lambda x}$. Figure 1 illustrates the reliability associated with a range of error rates for $n = 256$ iterations, and shows that the practical region of error rate is $\lambda < 0.05$.

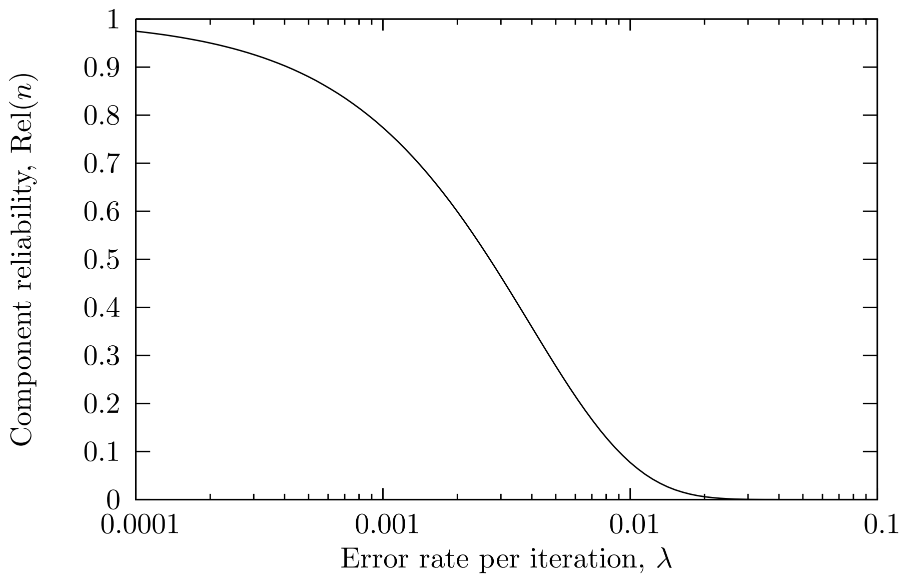

Figure 1: Reliability vs. error rate per iteration for $n = 256$ iterations

To describe the cost of computations and testing, the following notation is used. Let c_b be the average cost of an iteration of the base algorithm, e.g., a bare-bone scalar multiplication algorithm without any additional operations to enable error detection. For example, in the conventional double-and-add algorithm, $c_b = 3/2$ point operations on average, while for 2-NAF double-and-add algorithm, $c_b = 4/3$ on average. Let c be the cost of an iteration of an error-detecting algorithm. For example, in Algorithm 4, $c = 2$ point operations. Let c_v be the cost of the validation test used to detect the occurrence of errors in the state of the algorithm. As an example, the validation test in Algorithm 4 has $c_v = 4$ point operations. It should be mentioned that, while point operations are used uniformly hereafter as a cost measure, it is possible to use any other suitable computational cost unit, e.g. field or bit operations, without modifying the analysis.

3.2.2 A Statistical Cost Model

The overhead of error recovery using frequent validation and partial recomputation is defined as the increase in time required relative to a bare-bone scalar multiplication algorithm. In order to estimate this overhead, it is divided into three parts:

1. Iteration redundancy overhead: This part captures the cost of extra computations added in each of the original n iterations to allow for validation tests. It can be found by $n(c - c_b)$.
2. Extra iterations overhead: This part captures the cost of extra iterations, beyond the original n, that are required for recomputation in case of an error occurrence. It can be computed as $(\tilde{n} - n)c$, where $\tilde{n}$ denotes the total number of iterations including recomputations.
3. Testing overhead: Each block requires a validation test. This part captures that cost and can be computed as kc_v, where k denotes the total number of blocks.

Summing up these parts, we get the following function that describes the total overhead in point operations.

$$\begin{aligned} f_o(\tilde{n}, k) &= n(c - c_b) + (\tilde{n} - n)c + kc_v \\ &= \tilde{n}c - nc_b + kc_v \end{aligned} \tag{3}$$

This expression depends on two variables, $\tilde{n}$ and k. It follows by the linearity of expectations that

$$\mathrm{E}[f_o(\tilde{n}, k)] = \mathrm{E}[\tilde{N}]c - nc_b + \mathrm{E}[K]c_v \tag{4}$$

where $\tilde{N}$ and K denote the random variables describing $\tilde{n}$ and k, respectively.

At block boundaries, the state of the system can be completely described by the number of iterations done correctly so far, denoted by n_d, and the current block size, denoted by m_c. Let N be a random variable representing this state as follows.

$$N \in \{(n_d, m_c) : n_d, m_c \in \mathbb{Z}, n_d \in [0, n), m_c \in [m_l, m_h]\} \cup \{(n)\} \tag{5}$$

where (n) is the absorbing state representing the computation's completion. The total number of states is $n(m_h - m_l + 1) + 1$. Let the sequence $\{N_s\}$ be such that $N_0 = (0, m_{\text{init}})$ and

$$N_{s+1} = \begin{cases} (N_s(n_d), mDown(N_s(m_c))) & \text{if an error is detected} \\ (N_s(n_d) + N_s(m_c), mUp(N_s(m_c))) & \text{if } m_c < n - n_d\text{, error-free} \\ (n) & \text{if } m_c \geq n - n_d\text{, error-free} \end{cases} \tag{6}$$

where $N_s(n_d)$ and $N_s(m_c)$ are the current total of iterations done and the current block size at step s, respectively. Since the next state depends only on the current state and whether or not the current block has an error, this sequence is a Markov chain, and it can be used to find $\mathrm{E}[\tilde{N}]$ and $\mathrm{E}[K]$.

To describe this Markov chain we use an *Augmented Transition Matrix*, denoted by $\mathcal{A}(x)$, which is defined as follows.

$$\mathcal{A}(x) = \begin{pmatrix} \mathbf{A}(x) & \mathbf{A}^0(x) \\ \mathbf{0} & 1 \end{pmatrix} \tag{7}$$

where $\mathbf{A}(x)$ is a square matrix with dimension $n(m_h - m_l + 1)$, and $\mathbf{A}^0(x)$ is column vector satisfying $\mathbf{A}^0(1) + \mathbf{A}(1)\mathbf{1} = \mathbf{1}$. Here, $\mathbf{0}$ and $\mathbf{1}$ represent the all-0 and all-1 vectors, respectively, and the last row and column of $\mathcal{A}(x)$ correspond to the absorbing state (n). This matrix is a more general form of the conventional transition probability matrix in the sense that each transition probability is augmented with the corresponding transition distance or cost as the exponent of the variable x. More formally, each element $a_{ij}(x)$ of $\mathcal{A}(x)$ is defined as follows.

$$\begin{aligned} a_{ij}(x) &= p_{ij} x^{d_{ij}} \\ p_{ij} &= \Pr[N_{s+1} = (n_d, m_c)_j | N_s = (n_d, m_c)_i] \\ d_{ij} &= \min((m_c)_i, n - (n_d)_i) \end{aligned} \tag{8}$$

A conventional transition probability matrix can be obtained as $\mathcal{A}(1)$. The transition between any two states is determined by the adaptive policy and the error probability. All non-terminal states belong to one of two classes:

1. $(n_d, m_c) : m_c < n - n_d$. These states represent all blocks except for the last one in the computation. In this case, the next state is either $(n_d + m_c, mUp(m_c))$ with probability $e^{-\lambda m_c}$ (error-free), or $(n_d, mDown(m_c))$ with probability $1 - e^{-\lambda m_c}$ (error detected). In both cases, the power of x is the distance, which is m_c.

2. $(n_d, m_c) : m_c \geq n - n_d$. These states represent the (potentially) last block. The next state is either $(n_d, mDown(m_c))$ with probability $1 - e^{-\lambda(n-n_d)}$ (error detected), or the absorbing state (n), with probability $e^{-\lambda(n-n_d)}$ (error-free). In both cases, the power of x is the distance, which is $n - n_d$.

A probability vector, denoted by α, can be used to represent the initial conditions, e.g., the initial block size.

Finding E$[K]$

We now show how $\mathcal{A}(x)$ can be used to find the expected value of K, the number of blocks. In general, for a transition matrix $\mathcal{T}$ of the form

$$\mathcal{T} = \begin{pmatrix} \mathbf{T} & \mathbf{T}^0 \\ \mathbf{0} & 1 \end{pmatrix} \tag{9}$$

where $\mathbf{T}$ is a square matrix, $\mathbf{T}^0$ is a column vector and $\mathbf{T}^0 + \mathbf{T1} = \mathbf{1}$, the number of steps required to reach the absorbing state follows a *Discrete Phase-type distribution*, denoted by DPH$(\tau, \mathbf{T})$, where τ is a probability vector representing the initial distribution. Note that when $\mathcal{T} = \mathcal{A}(1)$ and $\tau = \alpha$, this distribution applies to K. It follows from the properties of the discrete phase-type distribution that

$$\Pr[K = k] = \alpha \mathbf{A}^{k-1}(1)\mathbf{A}^0 \tag{10}$$

and

$$\mathrm{E}[K] = \alpha(I - \mathbf{A}(1))^{-1}\mathbf{1} \tag{11}$$

where I is the appropriately-sized identity matrix.

Finding E$[\tilde{N}]$

The matrix $\mathcal{A}(x)$ can also be used to find the expected total number of iterations, E$[\tilde{N}]$. Recall that for each non-zero element $a_{ij}(x)$ of $\mathcal{A}(x)$, the coefficient represents the transition probability from state i to j while the exponent represents the distance, i.e., number of iterations in the transition.

So, to find the expected number of iterations between any two states i and j, we multiply the exponents by their respective coefficients and then divide by the sum of coefficients to normalize.

More formally, for a polynomial $a_{ij}(x) \neq 0$, we can find the expected number of iterations in the transition between states i and j as $\frac{a'_{ij}(1)}{a_{ij}(1)}$, where $a'_{ij}(1)$ is the first derivative of $a_{ij}(x)$ with respect to x evaluated at $x = 1$. This idea extends naturally to powers of $\mathcal{A}(x)$ to find the expected number of iterations over more than one block.

To find $\mathrm{E}[\tilde{N}]$, we find the expected number of iterations from the starting state to the absorbing state for all values of k and find their sum weighted by the probability of each respective k. More formally,

$$\begin{aligned} \mathrm{E}[\tilde{N}] &= \sum_{k\in K} \Pr[K=k]\mathrm{E}[\tilde{N}|K=k] \\ &= \sum_{k\in K} \Pr[K=k] \sum_{\tilde{n}\in\tilde{N}} \tilde{n}\Pr[\tilde{N}=\tilde{n}|K=k] \\ &= \sum_{k\in K} \Pr[K=k] \frac{(\frac{d}{dx}(\alpha\mathcal{A}^k)_{-1})(1)}{(\alpha\mathcal{A}^k)_{-1}(1)} \end{aligned} \tag{12}$$

where $(\alpha\mathcal{A}^k)_{-1}$ is the last element in the row vector $\alpha\mathcal{A}^k$. It is important to note that the last element in the last row of $\mathcal{A}(x)$ corresponding to $\Pr[N_{s+1} = (n)|N_s = (n)]$ should be set to 0 to prevent the accumulation of elements in the last column of $\mathcal{A}^k(x)$ between successive values of k. This reflects the fact that no more steps are taken beyond the absorbing state. Using (11) and (12), we can find the expected value of the total overhead in (3).

Note on computational complexity Finding $\mathrm{E}[\tilde{N}]$ requires raising $\mathcal{A}(x)$ to the power of all values of k, which will cause the number of nonzero elements to increase quickly. This can be mitigated by using α to reduce matrix multiplications to vector-matrix multiplications, e.g., computing $\alpha\mathcal{A}^3(x)$ as $(((\alpha\mathcal{A}(x))\mathcal{A}(x))\mathcal{A}(x))$ rather than $\alpha(\mathcal{A}^3(x))$. However, since elements are polynomials, this still allows the number of terms in each polynomial element to increase rapidly. As such, computing the powers of $\mathcal{A}(x)$ can become costly unless the number of states is limited. Here we give another representation of $\mathcal{A}(x)$ that significantly reduces the computational cost of finding $\mathrm{E}[\tilde{N}]$.

Note that in (12), for each k, we need to raise $\mathcal{A}(x)$ to the power of k in order to get a specific polynomial, denoted here by $a(x)$, which describes the k-steps transition from the starting state to the absorbing state. The average distance using k steps can then be found by $\frac{a'(1)}{a(1)}$. This value is multiplied by the probability of k and accumulated to get the average total number of iterations required. The key here is the computation of the value $\frac{a'(1)}{a(1)}$. Since $a(x)$ is a dot product, we can write $a(x) = \sum_i h_i(x)g_i(x)$ where $h(x)$ and $g(x)$ are row and column vectors, respectively. It follows that

$$a'(x) = \sum_i (h_i(x)g_i(x))' = \sum_i h'_i(x)g_i(x)+h_i(x)g'_i(x) = h'(x)g(x)+h(x)g'(x) \tag{13}$$

So, all we need to calculate $a(1)$ and $a'(1)$ is to know $h(1)$, $h'(1)$, $g(1)$ and $g'(1)$. It follows that the required powers of the polynomial matrix $\mathcal{A}(x)$ can be found using a pair of purely-numerical matrices, $\mathcal{A}(1)$ and $\mathcal{A}'(1)$. This reduces the computation cost significantly since it replaces each polynomial multiplication with three numerical multiplication and one addition. It also leads to significant saving in memory requirements as it avoids the rapid growth in the size of polynomial elements.

Short- and Long-term Behavior

In the preceding analysis, it was mentioned that the probability vector α can represents the initial distribution and can be used to select the initial block size by setting all elements to 0 except for the corresponding state, $(0, m_{\text{init}})$, which is set to 1. This effectively captures the transient behavior of the system and models the scenario where the initial block size is reset to m_{init} before starting each scalar multiplication.

However, it is possible not to reset the initial block size and keep the last used block size between completed computation. This has the advantage of maintaining the acquired information about the error rate and the suitable block size. To model this behavior, we need to find the long-term distribution for the block sizes and use it as α. We find this information as follows. First, we define M, which is the random variable that describes the current block size. Then, another Markov chain, denoted by $\{M_i\}$, is constructed which only describes the variation in block size, i.e., where states represent different block sizes within the allowed range. In particular, for $M_i = m$, The next state will be either $M_{i+1} = mUp(m)$ with probability $e^{-\lambda m}$, or $M_{i+1} = mDown(m)$ with probability $1 - e^{-\lambda m}$. Given the transition matrix of this chain, we need to find the associated stationary distribution. Such distribution, denoted by π, can be computed as the normalized left eigenvector of the transition matrix associated with the eigenvalue 1. Given π, we can set the initial state in α to reflect the stationary distribution, and

Table 1: Expected overhead results in point operations using the earlier model in [1] and the model in Section 3.2.2. Values between parentheses represent the percentage overhead relative to a bare-bone ECSM implementation.

λ		m	Exp. overhead, earlier model	Exp. overhead, (3)
0.05	m_{opt}	5	536 (140%)	539
	$m_{\text{r-opt}}$	3	607 (158%)	609.8
0.01	m_{opt}	13	289 (75%)	289.2
	$m_{\text{r-opt}}$	7	322 (84%)	323.4
0.001	m_{opt}	44	175 (46%)	175.5
	$m_{\text{r-opt}}$	11	228 (59%)	230.6

the estimated overhead will then be the expected long-term overhead.

3.3 Numerical Examples

3.3.1 Fixed-block Error Recovery Example

We illustrate the use of the model discussed earlier and validate it by comparing its results to the model developed earlier in [1]. To enable this comparison, we use Algorithm 3 as in [1] with similar parameters, i.e., $n = 256$, $c = 2$, $c_b = 3/2$ and $c_v = 4$. Two criteria were proposed in [1] to find an optimal block size. The more intuitive one is to select the block size, denoted by m_{opt}, to minimize the expected overhead. It is also possible to select the block size, denoted by $m_{\text{r-opt}}$, to minimize the required overhead to achieve a given reliability requirement, e.g., 99.99%. In this example, we use the optimal block sizes for various values of λ as computed in [1]. Table 1 compares the results obtained from the model described in this section to the earlier results. It can be seen from the table that the two models produce matching results, thus confirming the validity of the preceding analysis.

We also use simulation to validate the results the preceding analysis. For fixed-block error recovery, Table 2 shows some of the simulation results and compares them with the results obtained through analysis.

Table 2: Comparison of expected overhead using the model in Section 3.2.2 with the simulation results. The expected overhead from the simulation is shown with a 99% confidence interval. Also, the reliability of the expected overhead and the overhead at 99% reliability are shown

λ		m	Analysis	Simulation		
			E[OH]	E[OH]	Rel at E[OH]	OH at 99% Rel.
0.05	m_{opt}	5	539	540.5 $\pm$4.8	50%	686
	$m_{\text{r-opt}}$	3	609.8	609.6 $\pm$4.3	48%	702
0.01	m_{opt}	13	289.2	290.0 $\pm$4.3	50%	448
	$m_{\text{r-opt}}$	7	323.4	322.5 $\pm$2.5	52%	402
0.001	m_{opt}	44	175.5	174.9 $\pm$3.8	78%	336
	$m_{\text{r-opt}}$	11	230.6	230.7 $\pm$1.1	77%	302

3.3.2 Adaptive Error Recovery Example

We also demonstrate the results that can be achieved using adaptive error recovery. This example uses the same basic parameters used in the earlier example. Moreover, we have to select an adaptive policy, which amounts to setting the following parameters:

1. The block size bounds, m_l and m_h.
2. The block size update functions, $mUp()$ and $mDown()$.

As shown in [1], block sizes that are optimized for reliability are smaller than ones optimized for efficiency. Moreover, the former have an expected overhead that is close to the latter, and the former is less sensitive to the variation in the error rate. As such, it is sensible to give preference to smaller blocks as opposed to bigger blocks when setting the adaptive policy parameters. It is sensible as well to increase the block size (in the absence of error) slower than it is decreased (when an error is detected). This is because smaller blocks generally provide a better trade-off between efficiency and reliability.

Based on this argument, a good choice for the block size modification functions can be $mUp(m) = m + 1$ and $mDown(m) = \lfloor m/2 \rfloor$, as they are both easy to implement and satisfy the grow-slow/shrink-fast requirement. As for the range of block size values, we can select a range that includes the optimal block sizes for a range of values of λ, e.g., $[5, 40]$. In general, higher values of m should be avoided as they lead to less frequent validation tests, and potentially more expensive recomputations when errors occur.

Now that we have some values for these parameters, we can examine the results over a range of values for the initial block size, m_{init}, and the error rate, λ. Table 3 shows some analytical results for this example while Table 4 shows some simulation results. Figures 2, 3 and 4 show the expected overhead and the 99% reliability threshold for the short-term adaptive alternative compared to the fixed-block alternative for three values of λ over a range of initial block sizes. These tables and figures show that adaptive error recovery is generally more efficient than the fixed-block alternative except when the fixed block size happens to be near the optimal block size for a certain λ. Even in this case, the overhead of the adaptive alternative is not much higher. Moreover, while the 99% reliability threshold in the adaptive case varies with the initial block size, its variation is limited and indicates a low sensitivity to the choice of the initial block size.

On the other hand, Figure 5 compares the expected overhead of long-term adaptive error recovery to the overhead of the optimal fixed block size, i.e., one selected with the knowledge of λ, and shows that adaptive error recovery closely matches the overhead of optimized fixed-block error recovery in the long term. This indicates that, while the selection of m_{init} affects the expected overhead in the short term, the behavior in the long term will approach that of the best choice of m_{init} regardless of the actual initial block size. These results lead to the conclusion that the use of adaptive error recovery alleviates both the need to know the error rate in advance and the need to select a good value for m_{init}.

4 Error Recovery under a Variable Error Rate

In the previous section, the overhead associated with fixed-block and adaptive error recovery has been estimated using an analytical model and simulation under the assumption of a constant error rate. Both fixed-block and adaptive error recovery can perform relatively well under a constant error rate. In particular for the fixed-block alternative, the knowledge of the error rate is needed to optimize the block size for a particular error rate. It can be argued, however, that the constant error rate assumption does not capture the complexity of error occurrences in reality where the error rate is mostly either unknown or variable. In this section, we use simulation to explore the performance of both fixed-block and adaptive error recovery under randomly

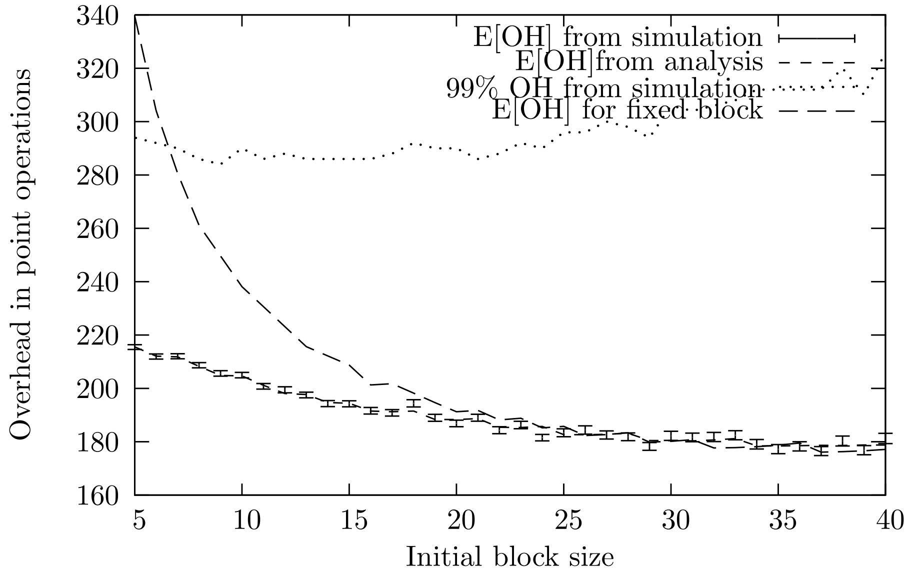

Figure 2: The expected overhead and 99% reliability threshold of adaptive error recovery compared to the fixed-block case for $\lambda = 0.001$. Simulation results are shown within 99% confidence interval.

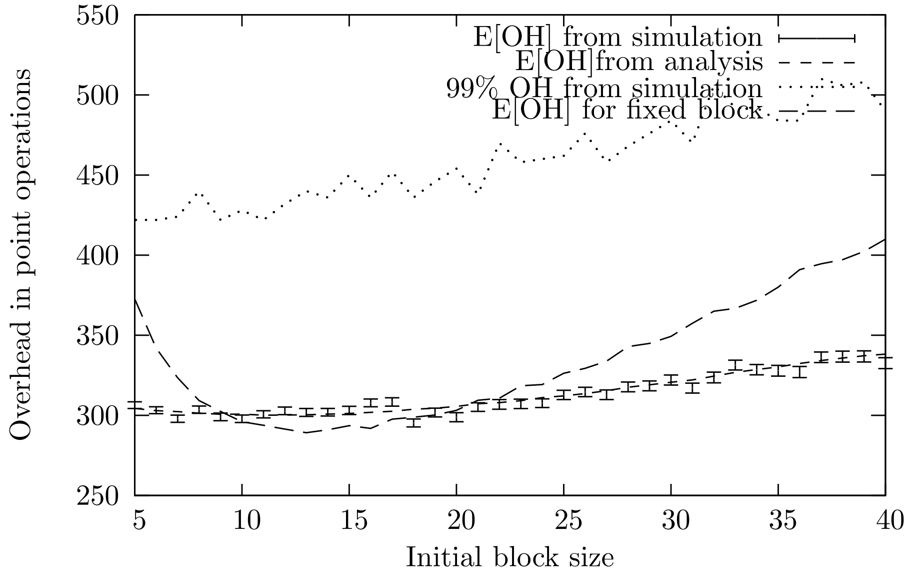

Figure 3: The expected overhead and 99% reliability threshold of adaptive error recovery compared to the fixed-block case for $\lambda = 0.01$.

Table 3: The expected overhead for adaptive error recovery for three values of λ. The expected overhead and expected block size are shown for three values of m_{init} in three error recovery scenarios, namely, fixed-block, and short- and long-term adaptive.

λ	m_{init}	Fixed-block	Adaptive, short-term		Adaptive, long-term	
		E[OH]	E[OH]	E[M]	E[OH]	E[M]
0.05	5	539.0	568.4	6.81	570.8	6.90
	15	845.8	587.3	7.23		
	40	3419.0	676.5	8.15		
0.01	5	372.6	304.3	10.13	302.5	11.83
	15	293.5	301.2	12.45		
	40	408.0	338.2	17.69		
0.001	5	339.6	215.6	13.00	181.3	28.33
	15	208.7	194.4	18.58		
	40	177.1	178.8	33.44		

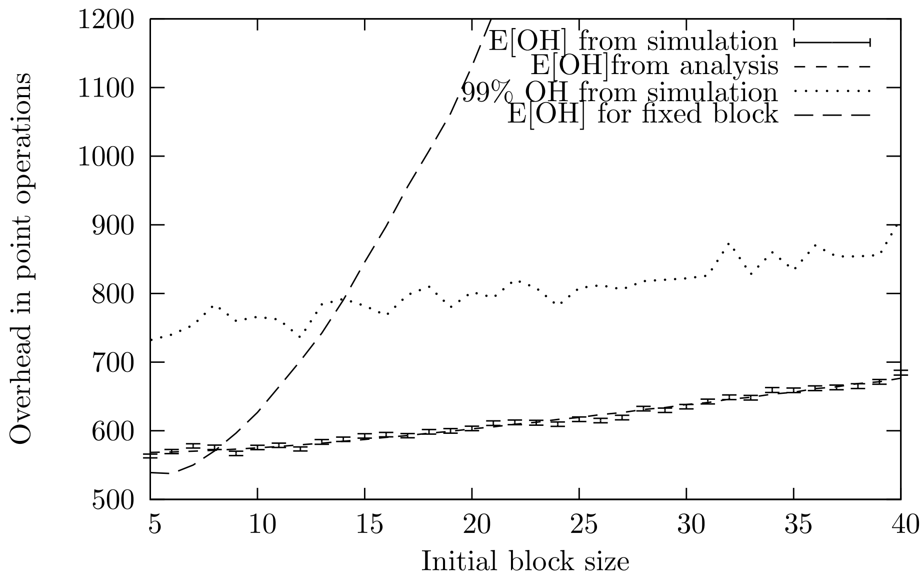

Figure 4: The expected overhead and 99% reliability threshold of adaptive error recovery compared to the fixed-block case for $\lambda = 0.05$.

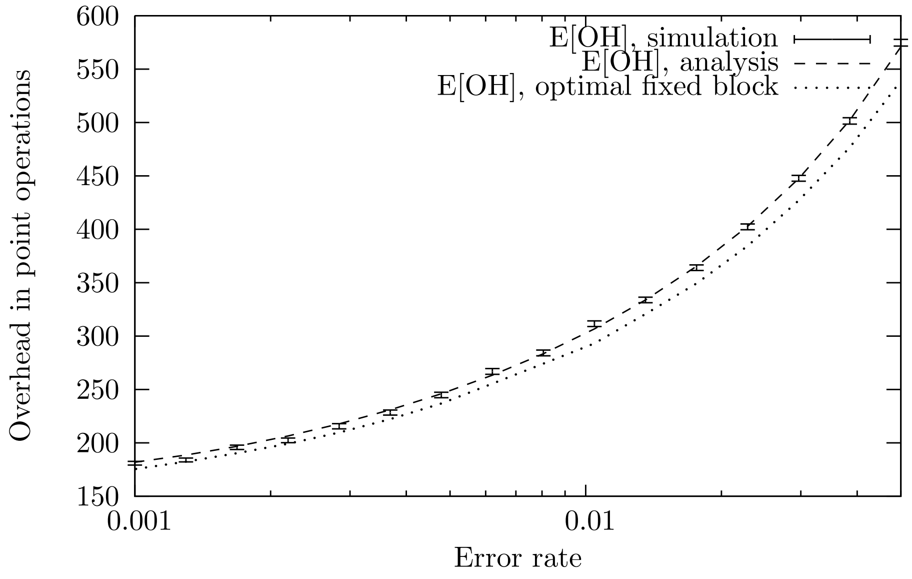

Figure 5: The expected overhead for long-term adaptive error recovery compared to the overhead of the optimal block size for a range of values of λ.

varying error rate.

4.1 A More General Error Model

For a more realistic error model, it is essential to address the possibility of burst errors, i.e., sudden increases in the perceived error rate that lead to the concentration of errors in a limited time period. This can be achieved by extending the error model discussed earlier using a variable error rate. In particular, a range of error rates, $[\lambda_{\text{low}}, \lambda_{\text{high}}]$, is selected and λ is picked randomly at block boundaries following a log-uniform distribution with parameters $\ln \lambda_{\text{low}}$ and $\ln \lambda_{\text{high}}$, i.e., $\ln \lambda \sim U(\ln \lambda_{\text{low}}, \ln \lambda_{\text{high}})$. The expected value of λ will then be $\frac{\lambda_{\text{high}} - \lambda_{\text{low}}}{\ln \lambda_{\text{high}} - \ln \lambda_{\text{low}}}$. This distribution has the property that most of its mass is closer to the lower end, but that occasionally results can come from the higher end. In effect, this represents the scenario where errors are generally rare with bursts of higher error rate.

4.2 A Numerical Example

We give an example to demonstrate the effectiveness of both fixed-block and adaptive error recovery under a variable error rate. We simulate both under the extended error model described earlier and gather information like the average overhead, the average block size and the overhead threshold that provides 99% reliability. The parameters used here are the same as earlier examples. For the range of λ, we use the conservative and relatively wide range $[0.001, 0.05]$. Error rates higher than 0.05 are excluded as they amount to 0 reliability of the original design, as illustrated in Figure 1, while error rates lower than 0.001 are low and can skew the simulation results to be more optimistic.

Table 5 shows the simulation results. As expected, adaptive error recovery is generally less sensitive to the value of m_{init}. Moreover, it can be more efficient and offer less expensive reliability than the fixed-block alternative. The only exception is when the fixed m_{init} is selected to be close to the unknown, optimal block size, which naturally results in less expected overhead. However, even then, adaptive blocks respond better to the variability of λ and give smaller reliability threshold.

Table 4: Simulation results for fixed-block and both short- and long-term adaptive error recovery under a constant error rate.

λ	m_{init}	Fixed-block		Adaptive, short-term				Adaptive, long-term			
		E[OH]	99% Rel OH	E[OH]	E[OH] Rel	99% Rel OH	E[M]	E[OH]	E[OH] Rel	99% Rel OH	E[M]
0.05	5	539	700	568	51%	752	6.8	572	51%	756	7
	15	835	1458	586	51%	776	7.2				
	40	3333	7608	676	50%	860	8.1				
0.01	5	372	434	304	52%	424	10.1	303	51%	448	12.7
	15	293	438	298	50%	436	12.5				
	40	405	912	331	50%	506	18.1				
0.001	5	339	364	215	77%	292	13	180	77%	316	30.6
	15	207	268	195	77%	288	18.6				
	40	176	324	179	77%	316	33.5				

Table 5: The simulation results for both fixed-block and adaptive error recovery starting from the same value of m_{init} under a variable λ in the range $[0.001, 0.05]$. In all runs, $\mathrm{E}[\lambda] = 0.0125$.

m_{init}	Fixed			Adaptive, short-term				Adaptive, long-term			
	E[OH]	Rel E[OH]	99% Rel OH	E[OH]	Rel E[OH]	99% Rel OH	E[M]	E[OH]	Rel E[OH]	99% Rel OH	E[M]
5	380	61%	448	319	48%	448	9.8	317	46%	464	11.9
15	307	61%	506	314	46%	462	11.9	315	47%	468	12
40	411	62%	912	348	45%	536	17.0	317	46%	466	11.9

Figure 6 shows the overhead for both short- and long-term adaptive error recovery compared to the fixed-block approach under a variable error rate. With the exception block sizes that happen to be close to optimal, adaptive blocks have less expected overhead than fixed ones. Figure 7 shows the 99% reliability overhead for short- and long-term adaptive error recovery compared to fixed blocks. Similar to the preceding figure, for all block sizes except for a small near-optimal range of fixed block sizes, adaptive error recovery offers the same reliability with significantly smaller expected overhead.

5 Effects of Adaptive Error Recovery on Security

Given the existence and proven practicality of a variety of side-channel attacks on the ECSM operation, it is essential to study the effect of frequent validation in general, and adaptive error recovery in particular, on the security of an ECSM implementation. The effects of frequent validation and partial recomputation on security have been addressed in [1]. In summary, the following issues were discussed:

1. Fault Analysis Attacks: The validation tests used in the preceding examples are known to be immune to a wide variety of fault attacks mounted by less-sophisticated attackers, i.e., attackers who do not have full control over the location and timing of injected faults. On the other hand, it is essential to avoid the single point of failure created by the validation tests being logical tests. This can be achieved using infective computation at the end of the computation to mask any faulty results.

2. Timing and Simple Power Analysis Attacks: Both fixed-block and adaptive error recovery introduce variability in the time required to complete the computation. However, this variability is dependent only on the location of errors and is independent of the secret information. Hence, if the underlying algorithm is immune to timing analysis, e.g., a double-and-add-always algorithm, then the use of frequent validation will not enable timing attacks.

3. Differential Power or Timing Analysis Attacks: Randomization can be used to counter differential attacks. Both fixed-block and adaptive error recovery schemes allow for the randomization of the whole computation, as well as the separate randomization of blocks, to avoid various types of differential attacks.

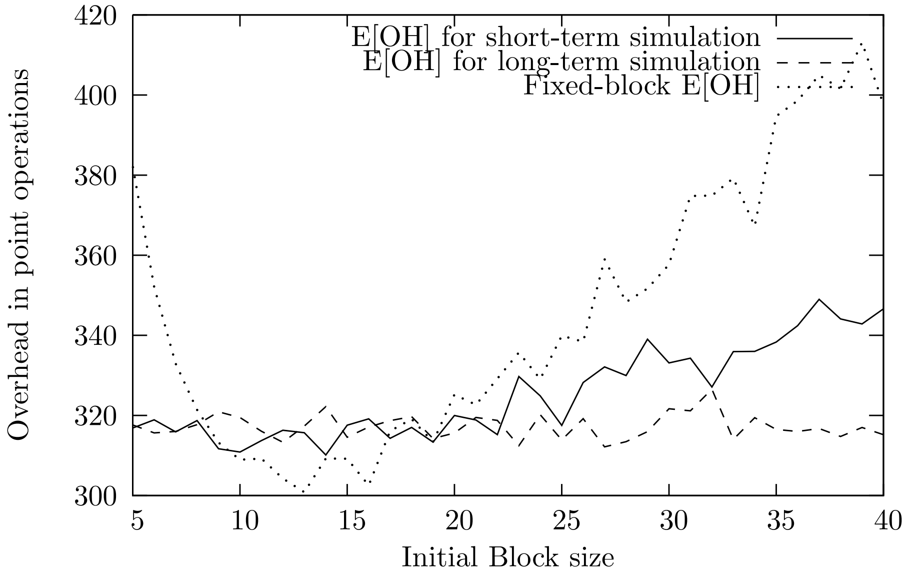

Figure 6: The expected overhead for adaptive compared to fixed-block for both short- and long-term

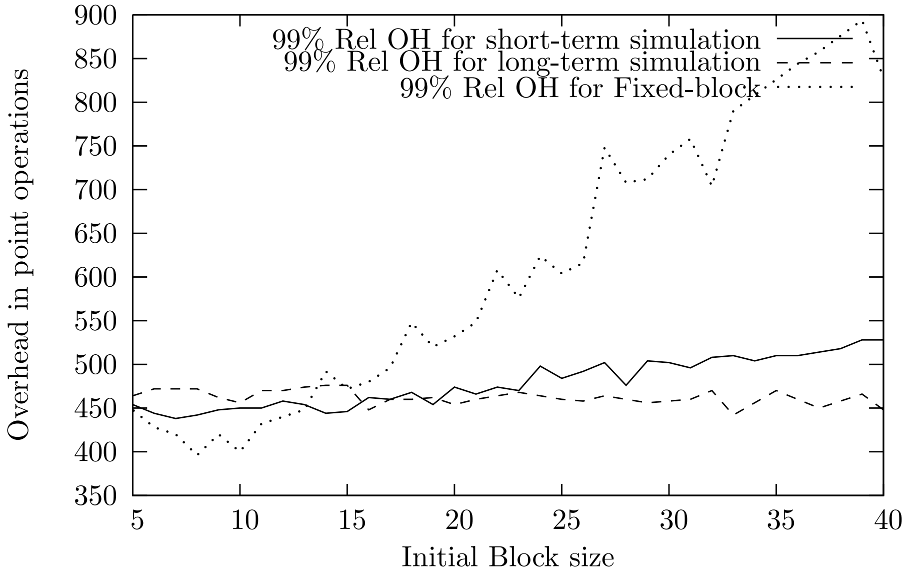

Figure 7: The 99% reliability overhead for adaptive compared to fixed-block for both short- and long-term

6 Conclusion

The advantage of using frequent validation with partial recomputation for error recovery in ECC implementations has been established in [1]. However, this advantage depends mainly on identifying a good trade-off between reliability and overhead, which requires some knowledge of the statistical model of errors. This can be overcome partially by selecting smaller block sizes that generally provide higher reliability while not increasing the overhead significantly.

In this work, we have introduced another approach to address this issue. Instead of fixing the block size to an optimal value, the block size is allowed to vary adaptively as a response to the occurrence of errors. We have shown, using an analytical model and using simulation, that this can give near-optimal reliability and efficiency while not requiring prior knowledge of the statistical parameters of errors. These results should motivate designers to incorporate adaptive error recovery in their designs.

References

[1] Abdulaziz Alkhoraidly and M. Anwar Hasan. Error detection and recovery for transient faults in elliptic curve cryptosystems. Technical Report CACR 2009-06, University of Waterloo, 2009. A revised version is under review.

[2] Adrian Antipa, Daniel R. L. Brown, Alfred Menezes, René Struik, and Scott A. Vanstone. Validation of elliptic curve public keys. In *PKC 2003: Public Key Cryptography*, LNCS 2567, pages 211–223. Springer-Verlag, 2003.

[3] 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.

[4] 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.

[5] Mathieu Ciet and Marc Joye. Elliptic curve cryptosystems in the presence of permanent and transient faults. *Des. Codes Cryptography*, 36(1):33–43, 2005.

[6] Henri Cohen, Gerhard Frey, and Roberto Avanzi. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press, 2006.

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

[8] Agustín Domínguez-Oviedo and M. Anwar Hasan. Error detection and fault tolerance in ecsm using input randomization. *Dependable and Secure Computing, IEEE Transactions on*, 6(3):175–187, July-Sept. 2009.

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

[10] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

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

[12] Victor S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *Lecture Notes in Computer Science; 218 on Crypto '85: Advances in Cryptology*, pages 417–426. Springer-Verlag, 1986.

[13] Peter L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[14] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[15] Sung-Ming Yen and Marc Joye. Checking before output may not be enough against fault-based cryptanalysis. *Computers, IEEE Transactions on*, 49(9):967–970, sep 2000.

[16] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sang-Jae Moon. RSA speedup with chinese remainder theorem immune against hardware fault cryptanalysis. *IEEE Trans. Comput.*, 52(4):461–472, 2003.