Built-In Self-Test for Modeled Faults

Mody Lempel

CENG Technical Report 94-37

Department of Electrical Engineering - Systems
University of Southern
Los Angeles, California 90089-2562
(213)740-4469

December 1994

Built-In Self-Test for Modeled Faults

by

Mody Lempel

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Science)

December 1994

Copyright 1994 Mody Lempel

Dedication

To my parents, my guiding light;

To Irit, *at lee ron.*

Acknowledgements

This thesis owes a great deal to Professors Melvin A. Breuer, Sandeep K. Gupta, Leonard M. Adelman, Douglas J. Ierardi and Lloyd R. Welch. I would like to thank you for your time, your wisdom and your patience. It was an honor and a pleasure learning from you.

This work was supported by the Advanced Research Projects Agency and monitored by the Federal Bureau of Investigation under Contract No. JFBI90092.

Contents

Dedication ii

Acknowledgements iii

List Of Tables vii

List Of Figures viii

Abstract ix

1 Introduction 1

1.1 Pattern generator design 4

1.2 Response analyzer design 8

2 Pattern generator design 12

2.1 The probability of 100% fault coverage with a random test sequence of length L 13

2.2 Detectability profile models 19

2.3 Experimental results on finding short pseudo random test sequences . 23

2.4 Discrete logarithms and LFSR sequences 27

2.5 The test embedding problem 29

2.6 Identifying hard faults 32

2.6.1 The heuristic 34

2.6.2 Probabilistic analysis of the heuristic 37
2.7 Experimental results 43
2.8 Conclusions 53
2.9 Miscellaneous issues 55
2.9.1 Multiple seeds 55
2.9.2 Lower fault coverage 56
2.9.3 Embedding subsets of test patterns 57
2.9.4 Using wider pattern generators 57
2.9.5 Different windows from one polynomial 58

3 Response analyzer design **59**

3.1 Bounds on the least degree non-factor of a set of polynomials 60
3.1.1 The worst case bounds 61
3.1.2 The expected bounds 65
3.2 Polynomial operations in $GF[2]$ 68
3.2.1 Polynomial multiplication, division and gcd 69
3.2.2 x^{2^m} modulo $b(x)$ and x^t modulo $b(x)$ 70
3.3 Finding a non-factor of smallest degree for a given set of polynomials 71
3.3.1 The product of all distinct factors of the same degree for a given polynomial 73
3.3.2 The number of all distinct factors, of the same degree, for a set of polynomials 76
3.3.3 Finding a non-factor 79
3.4 Practical scenarios 80
3.4.1 Finding a non-factor of a pre-specified degree 81
3.4.2 Finding a non-factor fast 82
3.4.3 Exhaustive search 83
3.5 Experimental results 84
3.5.1 Random selections based on the absolute bounds 84

3.5.2 Random selections based on the expected bounds 85
3.5.3 Experiments on benchmark circuits 86
3.6 Conclusions 87

Reference List **89**

Appendix A

The least prime in an arithmetic progression 93
A.1 General Bounds 93
A.2 Some evidence for the likelihood of $k < 2^n$ 95
A.3 Finding the Least Prime in the Arithmetic Progression $k2^n + 1$ 98

Appendix B

Generating irreducible polynomials 101

Appendix C

Factoring a product of distinct irreducible polynomials of the same degree 104

List Of Tables

2.1 Fault distribution and probability bounds for a test sequence of length L . . . 24

2.2 Probability bounds for a test sequence of length $2L$ . . . 25

2.3 Number of random sequences of length less than $(2)L$, that detected all faults . . . 26

2.4 Experimental results for some ISCAS85 combinational benchmarks. . . 44

2.5 Characteristics of synthesized circuits. . . . 45

2.6 RS results for the randomly testable synthesized circuits. . . . 46

2.7 RS results for the random resistant synthesized circuits. . . . 47

2.8 Hard fault identification for synthesized circuits. . . . 48

2.9 Embedding results for the synthesized circuits. . . . 49

2.10 $p_{nd}(el, t)$ values for $t = 2^j$, $l \leq j \leq l + 5$. . . . 49

2.11 Comparison of embedding results and random selection result. . . . 50

2.12 Distribution of simulation length vs. embedding length ratio. . . . 52

2.13 Results of additional RS experiments to get equal time comparisons. . 53

3.1 Number of primitive elements in $GF[2^m]$ and accumulated number of primitive elements in $GF[2^2] \ldots GF[2^m]$ . . . 64

3.2 Least prime p, of the form $\beta 2^m + 1$, with smallest generator α and 2^m-th root of unity ω. . . . 70

3.3 Summary of Results . . . 88

A.1 Smallest $1 \leq k \leq 100$ for which $k2^n + 1$ is prime, $0 \leq n \leq 206$ . . . 100

List Of Figures

1.1 A LFSR-based PG for a 4 input CUT. The feedback polynomial is $f(x) = x^4 + x + 1$. 3

1.2 A MISR-based RA for a 4 output CUT. The feedback polynomial is $f(x) = x^4 + x + 1$. 3

2.1 A LFSR with the feedback polynomial $f(x) = x^4 + x + 1$. 28

2.2 The procedure for the *windowing stage* of *EP* 31

2.3 The procedure for the second phase of identifying hard faults 35

3.1 Procedure $distinct_factors(h_i)$. Computes the product of all distinct factors of degree j, for $1 \leq j \leq u$, of the polynomial h. 74

3.2 Procedure $distinct_primitive(g_{i,j})$. Sifts out the non-primitive factors of $g_{i,j}$. 74

Abstract

Built-In Self-Test (*BIST*) is the capability of a circuit to test itself. The idea behind *BIST* is to create *pattern generators (PGs)* to generate patterns for the circuit and *response analyzers (RAs)* to *compact* the circuit response.

The *PG*s need to generate patterns that detect the *faults of interest* while keeping the length of the test sequence under user specified constraints.

The compaction function of the *RA*s is a *many-to-one* function, hence certain faulty responses might be mapped to the same *signature* as the good response. This is known as *aliasing.* When aliasing occurs, a faulty circuit is presumed to be fault free. The *RA* mechanism should minimize the aliasing probability.

Both the *PG* and *RA* mechanism should keep the hardware overhead to a minimum, thus reducing the area and delay penalty the circuit suffers.

Most circuits are too large to be tested as one entity, hence they are partitioned for test purposes. The partition is usually done at register boundaries. Thus, two popular mechanisms are a *linear feedback shift register (LFSR)* for the *PG* and a *multiple input shift register (MISR)* for the *RA*. The properties of these structures is determined by their *feedback polynomials.*

In this work we will focus on selecting feedback polynomials for *LFSR*-based *PG*s in order to achieve 100% fault coverage in minimum time and on selecting feedback polynomials for *MISR*-based *RA*s in order to achieve *zero-aliasing.* When a register acts as both a *PG* and a *RA*, we will select a polynomial that achieves both goals simultaneously.

The selection of a feedback polynomial for a *PG* is based on random selections for *randomly testable circuits* or on a guided search using the theory of *discrete logarithms* for *random resistant circuits.* For these circuits the set of faults is partitioned into *easy* and *hard* faults, the test patterns of the *hard* faults are generated, their position in the sequence generated by the LFSR is computed and a minimum length test sequence is found.

The selection of a *zero-aliasing* feedback polynomial for a *RA* is found by considering the error sequences for the faults of interest. A bound is computed that guarantees that a zero-aliasing polynomial of degree less than the bound exists. The factors of the error polynomials whose degrees are less than the bound are extracted and a polynomial that is a non-factor is searched for.

When the same polynomial is to serve both objectives, the polynomial chosen is the one that generates the shortest test sequence from a set of zero-aliasing polynomials.

Chapter 1

Introduction

Built-In Self-Test (BIST) is the capability of a circuit to test itself. The idea behind BIST is to create *pattern generators (PGs)* to generate test patterns for the circuit and *response analyzers (RAs)* to *compact* the circuit response to the inputs that are applied.

It is the responsibility of the PG to generate test patterns that detect all the *faults of interest.* The faults of interest depend on the *fault model* that is adopted. An example of a fault model is the *single stuck-at fault* for which the faults of interest are the non-equivalent single stuck-at faults. Under this model it is assumed that the *circuit under test (CUT)* has just one fault, and the fault is a single line which is either a constant "1" or a constant "0", independent of the circuit inputs. Other fault models include *bridging faults*, *open lines*, *transistor shorts* and *transistor stuck-on (off).* By knowing the fault model it is possible to simulate the effect of the fault, thereby to evaluate if a certain test pattern propagates the effect of the fault to an output, i.e. the output in the presence of the fault is different than the output of a fault free circuit. If one abandons the fault models and is interested in detecting any possible non-sequential fault, the PG must generate either an *exhaustive* test set in which all possible patterns are applied to the circuit, or a *pseudo exhaustive* test set in which all *logic cones* (a logic cone is the portion of logic that drives a single output bit) are tested exhaustively (this excludes faults that change the cone configuration of the circuit, e.g. certain bridging faults).

Once the PG mechanism is known, since it is deterministic and always initialized to the same state, the sequence of test patterns is also known. By simulation, the circuit response can be found. It is the responsibility of the RA to analyze the response and provide a *go/no-go* signal as to whether the CUT is fault free or faulty. The circuit response, which may consist of thousands of bits, is compacted into a *signature* which consists of only tens of bits. The compacting function is a *many-to-one* function and as a result some erroneous responses might be mapped to the same signature as the good response. This is known as *aliasing.* When aliasing occurs, a faulty circuit passes the test and is presumed to be fault free. When all erroneous responses are mapped to a different signature than the good response, we have *zero-aliasing.*

When designing a PG or a RA it is important that the design be effective and efficient. The effectiveness of a PG is measured in terms of the *fault coverage* it achieves, i.e. the percentage of faults of interest that are detected by the test sequence that is generated, and efficiency is measured in terms of test time, i.e. the length of the test sequence that needs to be applied in order to achieve the stated fault coverage. The effectiveness of a RA is measured in terms of the *aliasing probability*, i.e. the probability that a faulty circuit passes as fault free. It is also important that the design add minimal area and delay overhead to the circuit.

Testing a large circuit is done by partitioning the circuit into smaller subcircuits. Usually these subcircuits are combinational, hence they are bounded by registers. Thus most PG and RA designs are based on reconfiguring existing registers. Two popular mechanisms are the *linear feedback shift register (LFSR)* for the PG and the *multiple input shift register (MISR)* for the RA.

An example of a LFSR-based PG is shown in Figure 1.1. Number the cells of a k stage LFSR $D_0, D_1, \ldots, D_{k-1}$, with the feedback coming out of cell D_{k-1}. The feedback function is represented as a polynomial $f(x) = x^k + \sum_{i=0}^{k-1} f_i x^i$ and the feedback feeds cell D_i iff $f_i = 1$. The feedback polynomial of the LFSR in Figure 1.1 is $f(x) = x^4 + x + 1$. When the feedback polynomial is primitive, all $2^k - 1$ non-zero binary k-tuples are generated by the PG, hence it has the ability to generate all possible non-zero test patterns.

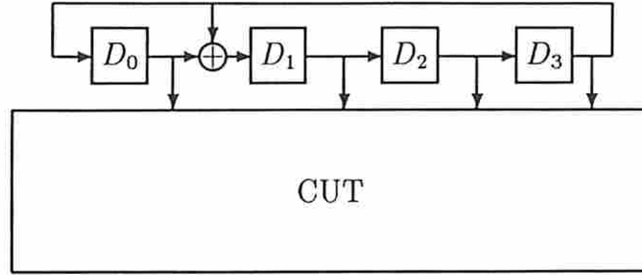

Figure 1.1: A LFSR-based PG for a 4 input CUT. The feedback polynomial is $f(x) = x^4 + x + 1$.

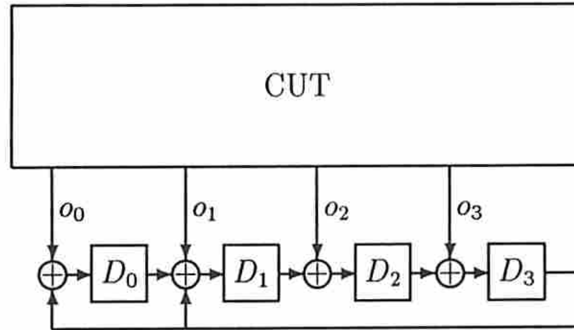

Figure 1.2: A MISR-based RA for a 4 output CUT. The feedback polynomial is $f(x) = x^4 + x + 1$.

An example of a MISR-based RA is shown in Figure 1.2. The difference between a MISR and a LFSR is that the CUT also feeds the stages of the shift registers. The compacting function of a MISR is polynomial division over $GF[2]$. The *effective output polynomial* is divided by the feedback polynomial. The signature is the remainder of the division. If the CUT has k outputs, it has k output sequences. Denote these sequences $o_0, o_1, \ldots, o_{k-1}$ where o_i feeds D_i. If the input sequence is of length n, then each o_i can be viewed as a polynomial $O_i = \sum_{j=0}^{n-1} o_{i,j} x^{n-1-j}$, where $o_{i,j}$ is the output of the i-th output at time j. The effective polynomial is then $O = \sum_{l=0}^{k-1} O_l x^l$.

The major drawback of a LFSR-based PG is that with random selections of the feedback polynomial and the initial state, it may take a long time for all the necessary pattern to be generated. Several designs have been proposed to alleviate this problem. The drawback of these schemes is the added area overhead over the LFSR design.

Several researchers have analyzed the aliasing probability of MISR-based RAs. This probability comes out to be 2^{-k} for a k-stage MISR. If the MISR is short (less than 20 stages), this probability might not be satisfactory. Several researchers proposed *zero-aliasing* designs. These designs can detect *any* error in the output sequence, but incur a high area penalty.

As a result of the partitioning into subcircuits being done at register boundaries, in many instances a register will serve as both a PG and a RA. The designs of non-LFSR PGs do not take into account the need for RA capability as well, and vice versa, hence the circuit will be subject to the overhead of both designs.

The purpose of this work is threefold. First, the design of LFSR-based PGs that achieve 100% fault coverage in minimum time. Second, the design of zero-aliasing MISR-based RAs for the set of modeled faults we are interested in detecting. These two tasks are achieved by proper selection of the feedback polynomials and seeds, hence the third aspect of this work is to select polynomials that achieve *both* goals simultaneously, thus freeing the circuit of the overhead of reconfigurable designs.

In the next two sections of this chapter we elaborate on past work regarding pattern generator and response analyzer design and outline our approach.

1.1 Pattern generator design

A n-stage LFSR with a primitive feedback polynomial generates a permutation of all the non-zero binary n-tuples. Changing the polynomial changes the permutation. The seed specifies the starting position for scanning the permutation. To achieve 100% detection of the faults of interest the LFSR must generate patterns until the set of patterns contains at least one test pattern for every fault. The problem is that the permutation might not contain a relatively short subsequence of test patterns for all the faults, and even if it does, it is usually not known where this short subsequence begins.

There are two approaches to deal with the problem of generating all the required test patterns. The first is by trying to estimate the number of test patterns required

to achieve 100% fault coverage. The second is by adding hardware to guarantee 100% detection with a small test set. While the first approach keeps the hardware at a minimum, it significantly adds to the required test time. The second approach results in a very short test time, but requires significant hardware overhead. We will first expand on the above approaches and then discuss our ideas for the design of LFSR-based PGs.

To estimate the number of patterns required to achieve 100% fault coverage, a number of authors [26] [39] [45] asked how many test vectors need be applied to achieve a given fault coverage with a specified degree of confidence. The answers depend on the *detectability profile* of the circuit, i.e. the number of test patterns for each fault. For example, to achieve 100% fault coverage when the probability of detection of the hardest fault is p, Savir and Bardell [39] suggest using a test sequence of length $11/p$. The main motivation behind this question is that it eliminates the need for test generation and for fault simulation. With the advances in test generation and fault simulation tools (not to mention platform speed), we believe this is not as important as it used to be. On the other hand, shorter test sequences for combinational subcircuits allow for shorter test sessions for the whole circuit and allow for easier synthesis of *zero-aliasing* response analyzers (Chapter 3).

As opposed to trying to detect all the faults of interest with one pseudo-random sequence, a second approach is to either use special hardware to generate small test sets, or to use a short pseudo-random sequence to detect most of the faults, and then use special hardware to detect the remaining faults. The use of non-linear feedback functions are suggested in [11] and [12]. Using this approach, one first creates a test set T that achieves 100% fault coverage and then synthesizes logic to generate the test patterns. In [11] the patterns in T are viewed as states of a finite state machine. Logic is built around the register such that the states of the register generate these patterns. In [12] patterns are added to T such that T can be ordered in a way that consecutive patterns differ in only one bit position. A ROM is used to store these positions. A related approach is that of weighted random patterns [46], [4], [27]. Logic is added to the register to affect the probability that each register cell outputs either a “1” or a “0”. The probabilities are based on the test patterns which detect the faults of interest. A store-and-generate approach is suggested in

[2]. Precomputed patterns are stored in a ROM. Each pattern is loaded into a LFSR and the LFSR proceeds to generate test patterns. After a certain number of patterns have been generated, the next pattern is loaded into the LFSR. This can be seen as a multiple seeding of the LFSR, which at the extreme can be used to detect all but a certain number of faults with the first seed, and the remaining seeds are patterns for the undetected faults. One implementation of this idea is suggested in [1]. A ROM, counter and additional linear logic is used to generate the patterns of T. The authors note that it is very unlikely that T will be a set for which their scheme will work. Another implementation is suggested in [20]. In this scheme not only is the PG reseeded, but with each seed there is a corresponding feedback polynomial. Decoding logic is used to reconfigure the feedback connections of the register, depending on the encoding of the polynomial. Along similar lines, [13] suggests to divide T into subsets of linearly independent patterns. The linear connections of the LFSR can be modified such that each connection generates a different subset of T. In [44] it is argued that one can do with just one such subset of T which includes the patterns required to detect the random resistant faults. Remaining faults will be detected by the succeeding patterns generated by the LFSR. A different scheme is suggested in [14], where the patterns are stored in a ROM and the outputs of the CUT act as addresses to the ROM.

All the above PG schemes cause additional area and delay overhead compared to a LFSR-based PG, although they reduce the size of the test set, in some cases considerably. Another drawback of some of these schemes is that changing the feedback function of the LFSR makes the compaction properties of the resulting circuits extremely difficult to analyze.

In this work we try to find *short one-seed* pseudo-random test sequences that achieve 100% fault coverage. The term *short* is relative to the probability of detecting the hardest fault of a circuit. If this probability is p, then *short* will mean a test length of $L = \frac{1}{p}$. By using just one seed the BIST control circuitry is kept at a minimum.

We first show that a pseudo-random test sequence of length at most $2L$ has a high probability of achieving 100% fault coverage. Thus, a random process of

selecting a random primitive polynomial and a random seed for a LFSR-based PG is likely to produce the desired test length. By simulating up to $2L$ test patterns, one knows whether the desired sequence is found or not. If not, another random selection is made. This scheme will be best suited for *randomly testable circuits*, i.e. those circuits with high values of p. For *random resistant circuits* we suggest a more sophisticated way of selecting the feedback polynomials and seeds that will produce shorter test sequences and require less computation time. We use the theory of *discrete logarithms* to *embed* a subset of test patterns in a LFSR sequence, from which we produce the test sequence for all faults.

Our algorithm is based on the fact that a pseudo-random test sequence that detecs the *random resistant faults* will, with very high probability, include test patterns for the *random testable faults* as well. By embedding test patterns only for the random resistant faults the complexity of the algorithm is greatly reduced. At the same time, it requires a method to differentiate between random resistant and random testable faults.

Our scheme puts a premium on reducing hardware and delay overhead, whereas previous schemes concentrate on either reducing test time at the expense of area and delay or vice versa. We believe that simplicity and functionality is more important, although within the realm of our approach we also minimize test time. In fact, when combined with our approach for zero-aliasing, it is crucial that test time be reduced as much as possible.

The applicability of these schemes is dependent on the computational effort one is willing to expend and on the time a test sequence is allowed to run.

The tests we embed can either be one pattern tests for non-sequential faults, or, using schemes such as in [43], two-pattern tests for sequential faults.

As the pattern sequence of LFSRs and one-dimensional *Cellular Automata (CA)* with the same primitive characteristic function are isomorphic, [42], our algorithms will also work for CAs.

1.2 Response analyzer design

The compacting function of a RA is a many-to-one function, hence the issue of aliasing must be dealt with.

Much work has been done on computing the *aliasing probability.* While different researchers considered different error models, all results showed the probability to converge to 2^{-k}, where k is the length of the RA. The first model to be considered was the *uniform error model*, in which all error sequences for a single output circuit were equally likely. Under this model it is straightforward, using polynomial division, to show that the aliasing probability is 2^{-k}. This error model is not very realistic. Williams et al. [47] considered the *independent error model* (also referred to as the *symmetric error model* [48] and the *Bernoulli model* [38]) for a single output circuit, where the probability for an erroneous output bit is p and the errors are independent of one another. They showed that the aliasing probability converges to 2^{-k} when the degree of the feedback polynomial is k, and that it converges faster for primitive polynomials. Using the same error model, Saxena et al. [38] computed upper bounds on the aliasing probability that are dependent on the length of the test sequence and the order of the feedback polynomial, L_c. For test lengths smaller than the order, the upper bound is $(1+\epsilon)/L_c$. For test lengths equal to L_c the upper bound is 1, and for lengths greater than L_c the bound is $2/(L_c+1)$. Xavier et al. used the *asymmetric error model* for single output circuits. In this model the probabilities of a $1 \to 0$ error and a $0 \to 1$ error are not necessarily the same, as in the symmetric error model. They used simulation data to show that this error model is more accurate than the symmetric error model. Their experiments tend to show that under this error model, the aliasing probability converges to 2^{-k}.

For multiple output circuits, Pradhan et al. [31] used the *q-ary symmetric error model* in which the probability of no error is p and all $2^k - 1$ error patterns (at time j) are equally likely with probability $\frac{1-p}{2^k-1}$. The aliasing probability under this model converges to 2^{-k}. Kameda et al. [22] show that as long as there are at least 2 different error patterns with probability greater than zero and the feedback polynomial is irreducible, the aliasing probability converges to 2^{-k}, independent of the probability distribution on the error patterns.

When trying to achieve zero-aliasing, there are two approaches. The first assumes that any error can occur, while the second considers only errors that result from a specific fault model.

There are two previous schemes that try to detect any deviation from the good circuit response. The first is by Gupta et al. [17]. In this scheme the RA is a LFSR and the compacting function is polynomial division of the good response by the feedback polynomial. The scheme requires the *quotient* of the good response to be periodic. This is achieved by proper selection of the LFSR feedback polynomial once the good response is known. They give a bound of $n/2$ on the length of the required register, for a sequence of length n. The second scheme is by Chakrabarty and Hayes [8]. They use non-linear logic to detect errors in the response. The number of memory cells in their RA is $\lceil \log n \rceil$ but they have no bound on the extra logic required to implement their scheme.

The compacting function of a MISR is polynomial division over $GF[2]$. The *effective output polynomial* is divided by the feedback polynomial. The signature is the remainder of the division. Our objective is to select a feedback polynomial for the compacting MISR, given a set of modeled faults, such that an erroneous response resulting from any modeled fault is mapped to a different signature than the good response. The major difference between our scheme and the aforementioned zero-aliasing schemes is that we target a specific set of possible faults and try to achieve zero-aliasing for the error sequences resulting only from these faults. We *do not* try to recognize all possible error sequences, mainly because most of them will *never* occur. The fault model lets us *focus* on the *probable* error sequences. As a result, we use less hardware than the aforementioned schemes.

A previous method for finding zero-aliasing feedback polynomials for modeled faults was presented by Pomeranz et al. [30]. Different heuristics for finding a zero-aliasing polynomial are suggested, but these heuristics will not necessarily find an irreducible or primitive polynomial, which is very important if the register is also to function as a PG. They do not give bounds on the resulting or necessary degrees of their zero-aliasing polynomials, nor do they present results on the computational complexity of their methods.

For a CUT with few outputs, the available register might be too short to achieve zero-aliasing. In this case we need to lengthen the register by adding memory elements. To keep the hardware overhead at a minimum, we want to add as few flip-flops as possible, hence we are interested in a feedback polynomial of smallest degree that achieves our objective. When a register is to serve both as a PG and a RA, it is advantageous to have the feedback polynomial of the same degree as the available register, hence we are interested in a feedback polynomial of a pre-specified degree. At times, we might want to find a feedback polynomial fast, even if the resulting MISR requires extra flip-flops over the optimum.

We assume the following test scenario. The input sequence to the CUT has been designed so that the effective output polynomial due to any target fault is different from the effective polynomial of the good response. Denote the effective polynomial of the good response by r, then the effective polynomial due to fault i can be represent as $r + h_i$. By the linearity of the remaindering operation, we get a different remainder for this erroneous polynomial iff h_i is not divisible by the feedback polynomial. We assume we are given the error polynomials for each of the target faults.

The problem we deal with is the following: given a set of polynomials $H = \{h_1, h_2, \ldots, h_{|H|}\}$ find a polynomial that is relatively prime to all the polynomials of H.

Such a polynomial will be referred to as a *non-factor* of H. If a non-factor is used as the feedback polynomial for the compacting MISR, zero-aliasing is achieved for the set of target faults.

In particular, for irreducible and primitive feedback polynomials we propose to find (1) upper bounds on the smallest degree zero-aliasing MISR; (2) based on the *expected* number of factors of a given degree for a given polynomial, we refine this upper bound to an *expected bound.* (3) procedures for selecting a zero-aliasing MISR with the smallest degree; (4) procedures for selecting a zero-aliasing MISR of a pre-specified degree (under the condition that one exists); and (5) procedures for fast selection of a zero-aliasing MISR.

The computational complexity, as well as the expected complexity, of all the proposed procedures is analyzed, based on the upper bounds for the smallest degree non-factor and the expected bounds.

Chapter 2

Pattern generator design

In this chapter we deal with the design of efficient and effective pattern generators based on linear feedback shift registers (LFSR). Their effectiveness is measured in terms of generating test patterns for all the faults of interest, and efficiency in terms of minimum test length (time). Both ends will be accomplished by proper selection of the feedback polynomials (configuration) and the initial seed (state) of the LFSR.

Our goal is to find *short one-seed* pseudo-random test sequences that achieve 100% fc. The term *short* is relative to the probability of detecting the hardest fault of a circuit. If this probability is p, then *short* will mean $L = \frac{1}{p}$. By using just one seed the BIST control circuitry is kept at a minimum. We first show that a pseudo-random test sequence of length at most $2L$ has a high probability of achieving 100% fc. Thus, a random process of selecting a random primitive polynomial and a random seed for a LFSR-based PG is likely to produce the desired test length. By simulating up to $2L$ test patterns, one knows whether the desired sequence is found or not. If not, another random selection is made. This scheme will be best suited for *randomly testable circuits*, i.e. those circuits with high values of p. For *random resistant circuits* we suggest a more sophisticated way of selecting the feedback polynomials and seeds that will produce shorter test sequences in less time. We use the theory of *discrete logarithms* to *embed* a subset of test patterns in a LFSR sequence, from which we produce the test sequence for all faults. The applicability of these schemes is dependent on the computational effort one is willing to expend and on the time a test sequence is allowed to run.

The tests we embed can either be one pattern tests for non-sequential faults, or, using schemes such as in [43], for two-pattern tests.

As the pattern sequence of LFSRs and one-dimensional *Cellular Automata* with the same primitive characteristic function are isomorphic, [42], our algorithms will also work for CAs.

The rest of this chapter is organized as follows. Throughout the chapter, n denotes the number of inputs to the circuit under test (CUT). In Section 2.1 we analyze the probability of detecting a fault having 2^{k+i} test patterns, given a test sequence of length $L = 2^{n-k}$ and we give lower and upper bounds on the probability of detecting all the faults of interest. In Section 2.2, we assume two different detectability profile models and derive the probability bound values for 100% fc for circuits that abide by these models. In Section 2.3, we find the actual detectability profile for some example circuits, derive the probability of drawing short test sequences and conduct random experiments which validate our analytic results. These sections provide the basis for our claim that short test sequences can be found in acceptable time constraints. We then proceed to introduce our procedures for selecting primitive feedback polynomials and seeds for the LFSRs. In Section 2.4 we introduce the notion of *discrete logarithms* and show its relation to the sequencing of patterns by a LFSR. In Section 2.5 we state the *test embedding problem* and propose our solution. Section 2.6 defines and characterizes faults we classify as *hard faults*, which are of major importance to our algorithm. In Section 2.7 we present experimental results. We present our conclusions in Section 2.8 and some miscellaneous issues in Section 2.9.

2.1 The probability of 100% fault coverage with a random test sequence of length L

Consider a combinational circuit with n inputs. In this section we address the following two questions.

Given a fault with $K = 2^k$ test patterns, out of $N = 2^n$ possible input patterns to the circuit, what is the probability that a random test sequence of length $L = 2^{n-k}$ does not detect the fault, assuming the patterns are drawn with no replacement? What is the probability that a random sequence of length L does not detect a fault with 2^{k+i} test patterns, where $i \geq -1$?

To answer these questions we define the function $p_{nd}(t, L)$ which is the probability that a sequence of length L does not detect a fault with t test patterns. Hence, we are looking for the values of $p_{nd}(t, L)$ when t equals 2^{k+i} for $i \geq -1$. The total number of sequences (with no importance to order) of length L is $\begin{pmatrix} N \\ L \end{pmatrix}$. Of those, the number of sequences that do not detect a given fault with t test patterns is $\begin{pmatrix} N-t \\ L \end{pmatrix}$. Hence,

$$p_{nd}(t, L) = \frac{\begin{pmatrix} N-t \\ L \end{pmatrix}}{\begin{pmatrix} N \\ L \end{pmatrix}}.$$

Writing this expression in factorial form

$$p_{nd}(t, L) = \frac{(N-t)!(N-L)!}{(N-t-L)!N!}.$$

After cancelling the appropriate terms

$$\begin{aligned}
p_{nd}(t, L) &= \frac{(N-L)(N-L-1)\cdots(N-L-t+1)}{N(N-1)\cdots(N-t+1)} \\
&= \left(\frac{N-L}{N}\right)\left(\frac{N-L-1}{N-1}\right)\cdots\left(\frac{N-L-t+1}{N-t+1}\right) \\
&= \left(1-\frac{L}{N}\right)\left(1-\frac{L}{N-1}\right)\left(1-\frac{L}{N-t+1}\right) && (2.1) \\
&< \left(1-\frac{L}{N}\right)^t. && (2.2)
\end{aligned}$$

For $t = 2^k = K$,

$$\begin{aligned} p_{nd}(2^k, L) &< \left(1 - \frac{L}{N}\right)^K \\ &= \left(1 - \frac{1}{K}\right)^K \\ &< \frac{1}{e}. \end{aligned}$$

For $t = 2^{k-1} = K/2$,

$$\begin{aligned} p_{nd}(2^{k-1}, L) &< \left[\left(1 - \frac{1}{K}\right)^K\right]^{\frac{1}{2}} \\ &< \left(\frac{1}{e}\right)^{\frac{1}{2}}. \end{aligned}$$

For $t = 2^{k+i} = K \cdot 2^i$, $i \geq 1$

$$\begin{aligned} p_{nd}(2^{k+i}, L) &< \left[\left(1 - \frac{1}{K}\right)^K\right]^{2^i} \\ &< \left(\frac{1}{e}\right)^{2^i}. \end{aligned}$$

In general, for $t = wK$

$$\begin{aligned} p_{nd}(t, L) &< \left[\left(1 - \frac{1}{K}\right)^K\right]^w \\ &< \left(\frac{1}{e}\right)^w. \end{aligned} \tag{2.3}$$

In most cases of interest to us, t will be (much) less than L. For these cases, this upper bound is tight. By Equation (2.1)

$$p_{nd}(t, L) > \left(1 - \frac{L}{N - t + 1}\right)^t$$

hence

$$p_{nd}(t,L) \;>\; \left(1 - \frac{1}{K - \frac{t-1}{L}}\right)^t$$

and by the assumption of $t < L$

$$p_{nd}(t,L) > \left(1 - \frac{1}{K-1}\right)^t.$$

For $t = wK$, this becomes

$$p_{nd}(t,L) \;>\; \left[\left(1 - \frac{1}{K-1}\right)^K\right]^w. \tag{2.4}$$

When $K = 32$

$$p_{nd}(t,L) > \left(0.9519\frac{1}{e}\right)^w$$

and for $K = 64$

$$p_{nd}(t,L) > \left(0.9762\frac{1}{e}\right)^w.$$

Combining Equations (2.3) and (2.4) under the assumption $t = wK < L$ results in

$$\left[\left(1 - \frac{1}{K-1}\right)^K\right]^w < p_{nd}(t,L) < \left[\left(1 - \frac{1}{K}\right)^K\right]^w$$

and as K increases, the bounds become tighter.

We defined the sequence length to be $L = \frac{N}{K}$. If the sequence length is doubled, the probability of missing a fault with t test patterns is squared. By Equation (2.2)

$$\begin{aligned} p_{nd}(t,2L) \;&<\; \left(1 - \frac{2L}{N}\right)^t \\ &= \left[\left(1 - \frac{1}{K/2}\right)^{K/2}\right]^{2w} \\ &< \left(\frac{1}{e}\right)^{2w}. \end{aligned}$$

Similarly, if the test sequence length is halved $(L/2)$ the probability is the square root of its value for a sequence of length L.

Using the values of $p_{nd}(2^{k+i}, L)$, we would like to bound from above and below the probability that a sequence of length L detects all the faults of interest. We do this by taking for each value of t the closest power of 2 greater or equal to t and the closest power of 2 less than or equal to t. By considering the power of 2 greater than t we derive the upper bounds, and by considering the power of 2 less than t we derive the lower bounds.

Let $F = \{f_1, f_2, \ldots, f_s\}$ be the set of all the faults of interest in the CUT. Let t_i be the number of tests for fault f_i and let $k_i = \lceil \log t_i \rceil$. Let $k_{min} = \min_i\{k_i\}$ and $k_{max} = \max_i\{k_i\}$. Group the faults into subgroups $C_{k_{min}}, C_{k_{min}+1}, \ldots, C_{k_{max}}$, where $f_i \in C_j$ iff $k_i = j$.

Let $k = k_{min}$. We are interested in finding the probability that a random sequence of length L detects all the faults of F.

Let p_j be the probability that a fault in C_j is not detected by the sequence. Since a fault in C_j has between 2^{j-1} and 2^j test patterns, the probability p_j is lower bound by $p_j \geq p_{nd}(2^j, L)$. We know that $\frac{1}{e^{2^{j-k}}} > p_{nd}(2^j, L)$. The value of p_j is either greater than $\frac{1}{e^{2^{j-k}}}$ or less than or equal to $\frac{1}{e^{2^{j-k}}}$. Our goal is to get a *pessimistic* upper bound on the probability of detection, hence we will assume that

$$p_j > \frac{1}{e^{2^{j-k}}}.$$

The probability, q_j, that a fault in C_j is detected by the sequence is upper bound, by our above assumption, by

$$q_j < 1 - \frac{1}{e^{2^{j-k}}}.$$

Notice that without our assumption, the upper bound on q_j can be greater. Hence, the probability, q, that a random sequence of length L detects *all* the faults of F is upper bound by

$$q \ < \ \prod_{j=k}^{k_{max}} \left(1 - \frac{1}{e^{2^{j-k}}}\right)^{|C_j|}.$$

In the above expression we assume that the detectabilities of any two faults are independent.

Similarly, we can define $d_i = \lfloor \log t_i \rfloor$ and the subgroups $\{F_j\}$ where $f_i \in F_j$ iff $d_i = j$. Let $d = \min_i\{d_i\}$ and $d_{max} = \max_i\{d_i\}$, then d is either k or $k-1$ and d_{max} is either k_{max} or $k_{max}-1$. The probability r_j that a fault in F_j is not detected is upper bound by

$$r_j < \frac{1}{e^{2^{j-k}}}$$

hence the probability, s_j, that a fault in F_j is detected is lower bound by

$$s_j > 1 - \frac{1}{e^{2^{j-k}}}.$$

Thus, the probability q of detecting all the faults of interest is bound by

$$\prod_{j=d}^{d_{max}} \left(1 - \frac{1}{e^{2^{j-k}}}\right)^{|F_j|} < q < \prod_{j=k}^{k_{max}} \left(1 - \frac{1}{e^{2^{j-k}}}\right)^{|C_j|}. \tag{2.5}$$

The actual value of q will be closer to the upper (lower) bound when for most of the harder faults the number of test patterns is closer to the power of 2 from above (below).

The super-exponential decrease in the probability of not detecting a fault, as we move from C_j to C_{j+1}, allows us to consider only the first few subgroups of faults when estimating the probability q. This is an analytic explanation to a similar observation made in [39]. To see this, consider the following question. How many faults x_j, with 2^{j+1} test patterns are needed such that the product of their detection probability equals the detection probability of a single fault with 2^j test patterns? This can be written as

$$\begin{aligned} \left(1 - \frac{1}{e^{2^{(j+1)-k}}}\right)^{x_j} &= \left(1 - \frac{1}{e^{2^{j-k}}}\right) \quad \Rightarrow \\ x_j &= \frac{\ln\left(1 - \frac{1}{e^{2^{j-k}}}\right)}{\ln\left(1 - \frac{1}{e^{2^{j+1-k}}}\right)}. \end{aligned}$$

Substituting y for $e^{2^{j-k}}$ we get

$$x_j = \frac{\ln\left(1 - \frac{1}{y}\right)}{\ln\left[\left(1 - \frac{1}{y}\right)\left(1 + \frac{1}{y}\right)\right]} = \frac{\ln\left(1 - \frac{1}{y}\right)}{\ln\left(1 - \frac{1}{y}\right) + \ln\left(1 + \frac{1}{y}\right)}. \tag{2.6}$$

The power series expansion of $\ln(1+z)$, where $-1 < z < 1$ is

$$z - \frac{1}{2}z^2 + \frac{1}{3}z^3 - \frac{1}{4}z^4 - \ldots.$$

Substituting z for $\frac{1}{y}$ in Equation (2.6), and expanding to the first $2M$ terms, we get

$$\begin{aligned} x_j &= \frac{\sum_{i=1}^{2M} \frac{z^i}{i}}{\sum_{i=1}^{2M} \frac{z^i}{i} + \sum_{i=1}^{2M} (-1)^i \frac{z^i}{i}} \\ &= \frac{\sum_{i=1}^{2M} \frac{z^i}{i}}{\sum_{i=1}^{M} \frac{z^{2i}}{i}} \\ &= \frac{1}{z} \times \frac{\left[z + \frac{z^2}{2} + \frac{z^3}{3} + \frac{z^4}{4} + \ldots\right]}{\left[z + \frac{z^3}{2} + \frac{z^5}{3} + \frac{z^7}{4} + \ldots\right]} \\ &> \frac{1}{z} = e^{2^{j-k}}. \end{aligned}$$

For $j - k = -1, 0, 1, 2, 3$ and 4, the values of x_j are greater than $2(> e^{2^{-1}})$, $3(> e)$, $7(> e^2)$, $55(> e^{2^2})$, $2981(> e^{2^3})$, and $8886114(> e^{2^4})$ respectively. This means that when computing the bounds on q, the significance of one fault with 2^{k+4} test patterns to the probability of success is greater than the contribution of e^{16} faults with 2^{k+5} test patterns. Equivalently, the probability of detecting a single fault with 2^{k+4} test patterns is less than detecting e^{16} faults with 2^{k+5} test patterns. Thus, the faults in the subgroups $C_{d+5}, \ldots, C_{d_{max}}$ and $F_{k+5}, \ldots, F_{k_{max}}$ do not have to be taken into account when analyzing the probability of success.

2.2 Detectability profile models

The actual value of the probability q depends on the distribution of faults in the different subgroups. To get an idea of this value as a function of the number of faults in C_k and F_d, we assume two detectability profile models, both placing an emphasis on the first few subgroups. These models will let us approximate bounds on the probability q without considering the detectability profile, but only the total

number of faults and the number of faults in F_d and C_k. We later use the profiles of actual circuits and the results demonstrate that our two models are very pessimistic, i.e. we should expect better results from actual circuit distributions than from the model approximations.

The first model we consider is the exponential model. In this model we assume $d = k - 1$ and the following distribution:

$$\begin{array}{rclcrcl}
|F_d| & = & u & , & |C_k| & = & v \\
|F_{d+1}| & = & ue^{\frac{1}{2}} & , & |C_{k+1}| & = & ve \\
|F_{d+2}| & = & |F_{d+1}|e & , & |C_{k+2}| & = & |C_{k+1}|e^2 \\
|F_{d+3}| & = & |F_{d+2}|e^2 & , & |C_{k+3}| & = & |C_{k+2}|e^4 \\
|F_{d+4}| & = & |F_{d+3}|e^4 & , & |C_{k+4}| & = & |C_{k+3}|e^8 \\
|F_{d+5}| & = & |F_{d+4}|e^8 & , & |C_{k+5}| & = & |C_{k+4}|e^{16}.
\end{array}$$

The idea behind this model is that the probability of detecting all the faults in F_{d+i} (C_{k+i}) is the same as detecting all the faults in F_d (C_k). Using this model, we get

$$\begin{array}{rcl}
\left(1-\left(\frac{1}{e}\right)^{2^{-1}}\right)^{|F_d|} & \approx & \left(1-\left(\frac{1}{e}\right)^{2^{0}}\right)^{|F_{d+1}|} \\
\left(1-\left(\frac{1}{e}\right)^{2^{0}}\right)^{|F_{d+1}|} & \approx & \left(1-\left(\frac{1}{e}\right)^{2^{1}}\right)^{|F_{d+2}|} \\
& \vdots & \\
\left(1-\left(\frac{1}{e}\right)^{2^{8}}\right)^{|F_{d+4}|} & \approx & \left(1-\left(\frac{1}{e}\right)^{2^{16}}\right)^{|F_{d+5}|}.
\end{array}$$

The same is true for the subgroups $\{C_i\}$. Thus, we can approximate q with the bounds

$$\left(\left(1-\left(\frac{1}{e}\right)^{\frac{1}{2}}\right)^{|F_d|}\right)^{d_{max}-d} < q < \left(\left(1-\frac{1}{e}\right)^{|C_k|}\right)^{k_{max}-k}.$$

The values of $k_{max} - k$ and $d_{max} - d$ can be bound as follows. The number of faults in each of the subgroups C_{k+i} $(k + i < k_{max})$ is ve^{2^i-1}. The last subgroup, $C_{k_{max}}$, may not be completely full, hence

$$|F| < ve^{2^{k_{max}-k}}$$

thus,

$$\begin{aligned} \ln|F| &< \ln v + 2^{k_{max}-k} \qquad \Rightarrow \\ k_{max} - k &> \log(\ln|F| - \ln v). \end{aligned}$$

Similarly, the subgroup $F_{d_{max}-1}$ is full, hence

$$|F| > ue^{\frac{2^{d_{max}-d-1}-1}{2}}$$

and

$$d_{max} - d < \log(2(\ln|F| - \ln u) + 1) + 1$$

hence

$$\left(1 - \left(\tfrac{1}{e}\right)^{\frac{1}{2}}\right)^{u \cdot log(2(\ln|F| - \ln u)+1)+1)} < q < \left(1 - \tfrac{1}{e}\right)^{v \cdot log(\ln|F| - \ln v)}.$$

Setting, for example, u and v to equal 10, and $|F| = 5000$, we get

$$9 \cdot 10^{-20} < q < 6.61 \cdot 10^{-6}.$$

By doubling the test length the probability bounds become

$$4.34 \cdot 10^{-6} < q < 2.28 \cdot 10^{-2}.$$

The second distribution is the linear distribution, in which

$$\begin{aligned}
|F_d| &= u &,\quad |C_k| &= v \\
|F_{d+1}| &= e^{\frac{1}{2}}u &,\quad |C_{k+1}| &= ev \\
|F_{d+2}| &= e^{\frac{1}{2}}|F_{d+1}| &,\quad |C_{k+2}| &= e|C_{k+1}| \\
|F_{d+3}| &= e|F_{d+2}| &,\quad |C_{k+3}| &= e|C_{k+2}| \\
|F_{d+4}| &= e|F_{d+3}| &,\quad |C_{k+4}| &= e|C_{k+3}| \\
|F_{d+5}| &= e|F_{d+4}| &,\quad |C_{k+5}| &= e|C_{k+4}|.
\end{aligned}$$

Using this distribution

$$|F| = v \sum_{i=0}^{k_{max}-k} e^i.$$

With this distribution, the product of the probability of detecting all the faults in F_{d+2+i} $(0 \leq i \leq 3)$ is (much) greater than the probability of detecting one fault in $F_{d+2+i-1}$. The probability of detecting all the faults in F_{d+1} is greater than the probability of detecting all the faults in F_d. The same applies to the faults in the subgroups $\{C_j\}$. Thus, we can approximate the bound on the probability of detecting all the faults by

$$\left(1 - \left(\frac{1}{e}\right)^{\frac{1}{2}}\right)^{2u} < q < \left(1 - \frac{1}{e}\right)^{2v}. \tag{2.7}$$

Assuming $u = v = 10$, $|F|$ is irrelevant in this case,

$$7.91 \cdot 10^{-9} < q < 10^{-4}.$$

If we double the length of the test sequence, the bounds become

$$10^{-4} < q < 5.45 \cdot 10^{-2}.$$

The upper bounds, for both models, on the probability that a random test sequence of length $2L$ achieves 100% fc is better than $\frac{1}{50}$. These results, of course, are dependent on our choice of values for u, v and F. Irrespective of these values is the effect that doubling the test length has on the probability of 100% fc. This effect is

a result of the exponential decrease in the probability of not detecting a fault. While these two detectability profiles seem artificial, experiment circuits show them to be very pessimistic, i.e. *actual profiles give rise to detection probabilities that are better than the models' bounds.*

2.3 Experimental results on finding short pseudo random test sequences

We synthesized 13 circuits from the Berkeley [6] benchmarks as multilevel circuits. For each circuit we found its k value. Using a modified version of the ATALANTA [18] test generation system, we generated a list of all the non-equivalent faults of the circuit and proceeded to generate all possible test patterns for each fault. If the pattern count exceeded 2^{k+5}, we discarded the fault and stopped the test generation procedure for the fault. Otherwise, we recorded the number of test patterns for the fault. Having iterated through all the faults, we found the respective sizes of the subgroups $\{F_j\}$ and $\{C_j\}$. We then used Equation (2.5) to compute upper and lower bounds on the probability of finding a test sequence of length $L = 2^{n-k}$ which detects all the faults. These results are presented in Table 2.1.

The first row for each circuit is the number of faults in the subgroups F_{k-1} through F_{k+5} and the second row is the number of faults in the subgroups C_{k-1} through C_{k+5}. The column labeled q in each row is the bound derived from the row using Equation (2.5). In the first row is the lower bound on q and in the second the upper bound. The column labeled *lin. bnd.* represents the bounds derived using the linear detectability profile model (Equation (2.7)). The values of u and v, the number of faults in F_{k-1} and C_k, respectively, are taken from their entries in the table. When $u = 0$, as is the case for the first eight circuits, we cannot calculate the lower bound, hence the entry is left blank. Notice that only for circuit *in5*, where the number of faults in C_k was only 1, did the model give a bound that was higher than the one given by Equation (2.5). We also computed the bounds on the probability that a sequence of length $2L$ will detect all the faults. The results are in Table 2.2. Having computed the probability bounds, we conduct 100 experiments (for circuit

Table 2.1: Fault distribution and probability bounds for a test sequence of length L

circuit	$k-1$	k	$k+1$	$k+2$	$k+3$	$k+4$	$k+5$	q	*lin. bnd.*
bc0	0	10	18	9	21	30	274	0.00063	
	0	10	14	10	17	30	281	0.0011	0.0001
b3	0	3	6	20	18	33	138	0.0725	
	0	3	3	12	16	20	164	0.1301	0.0638
chkn	0	5	18	37	30	31	127	0.0037	
	0	4	5	30	34	23	152	0.0438	0.0255
cps	0	25	33	39	172	131	269	$4 * 10^{-8}$	
	0	25	33	35	163	101	312	$4.2 * 10^{-8}$	$1.1 * 10^{-10}$
exep	0	12	27	42	54	80	75	$3.6 * 10^{-5}$	
	0	12	22	37	42	72	105	$8.3 * 10^{-5}$	$1.6 * 10^{-5}$
in3	0	3	7	9	2	2	193	0.0772	
	0	3	3	5	8	2	195	0.1485	0.0638
in4	0	3	6	22	25	32	158	0.1078	
	0	3	3	12	22	25	181	0.1298	0.0638
in5	0	1	14	46	55	49	56	0.0346	
	0	1	12	32	42	39	95	0.0602	0.4
in7	0	5	0	0	7	9	77	0.1007	
	0	5	0	0	7	2	84	0.1007	0.01
vg2	5	9	1	9	9	0	33	0.0001	$8.9 * 10^{-5}$
	0	12	3	0	9	9	33	0.0026	$1.6 * 10^{-5}$
vtx1	8	12	0	9	15	17	12	$2 * 10^{-6}$	$3.3 * 10^{-7}$
	0	16	4	0	9	15	29	$3.6 * 10^{-4}$	$4.2 * 10^{-7}$
x1dn	8	12	0	9	15	17	12	$2 * 10^{-6}$	$3.3 * 10^{-7}$
	0	16	4	0	9	15	29	$3.6 * 10^{-4}$	$4.2 * 10^{-7}$
x9dn	8	6	18	20	5	9	17	$1.8 * 10^{-6}$	$3.3 * 10^{-7}$
	0	14	2	27	9	5	26	$7.3 * 10^{-4}$	$2.6 * 10^{-6}$

Table 2.2: Probability bounds for a test sequence of length $2L$

circuit	lower bound	upper bound
bc0	0.167	0.18
b3	0.5747	0.6091
chkn	0.3422	0.5045
cps	0.0141	0.0142
exep	0.1045	0.1149
in3	0.3207	0.6106
in4	0.5743	0.6091
in5	0.6573	0.6852
in7	0.4833	0.4833
vg2	0.0267	0.1650
vtx1	0.0044	0.0907
x1dn	0.0044	0.0907
x9dn	0.00759	0.1247

chkn only 50 experiments were conducted) of random selections of polynomials and seeds to produce 100 test sequences. We ran each sequence for at most $2L$ patterns (for circuit *chkn* at most $1.2L$), stopping whenever 100% detection was achieved. We recorded the number of random sequences of length at most L and of length between L and $2L$ that detected all the faults. The results are in Table 2.3. The first column shows the expected number of sequences of length at most L that detected all faults. These numbers are based on the probability values from Table 2.1. The second column shows the actual number of such sequences. Column three shows the number of sequences of length greater than L and at most $2L$ that detected all faults. Column four gives the total number of sequences of length at most $2L$ that detected all faults and column five gives the expected number of such sequences, based on the probability values from Table 2.2. For 24 cases (omitting circuit *chkn*), we have both expected and actual results. For 11 of the 24 cases, the actual results were in the expected range. Omitting the 5 cases in which expected and actual results were 0, of the remaining 6 cases, in 5 the actual result was closer to the higher expected value and in 1 case it was closer to the lower expected value. Of the 13 cases in which the actual value was not in the expected range, in 10 cases the actual results were higher than the high end of the expected range.

Table 2.3: Number of random sequences of length less than (2)L, that detected all faults

circuit	expc$\leq L$	$\leq L$	$L <, \leq 2L$	$\leq 2L$	expc$\leq 2L$
bc0	0	1	23	24	16-18
b3	7-13	12	49	61	57-61
chkn	0-4	1	2	*	34-50
cps	0	0	36	36	1
exep	0	0	14	14	10-12
in3	7-14	0	50	50	32-61
in4	10-13	17	47	64	57-61
in5	3-6	8	64	72	66-69
in7	10	9	37	46	48
vg2	0	0	18	18	2-17
vtx1	0	0	6	6	0-9
x1dn	0	0	2	2	0-9
x9dn	0	2	10	12	0-12

Our first conclusion, based on the empirical results, is that the probability bounds given by Equation (2.5) are fairly accurate. The fact that the actual results were usually in the high end of the expected range can be explained by the fact that (1) we used pessimistic assumptions to derive Equation (2.5) and (2) certain correlations between the detectability of some faults may exist but were not considered.

Our second conclusion from both the analytic and empirical results is that the bounds given by the linear distribution model are overly pessimistic and Equation (2.5) will give more optimistic results.

Given that the actual results tended to be in the high end of the expected range, we conclude that with only a few random selections sequences of length at most $2L$ can be found that produce 100% fc.

2.4 Discrete logarithms and LFSR sequences

Up to this point it was shown that the probability that a pseudo-random test sequence of length at most $2L$ achieves 100% fc is typically greater than $\frac{1}{50}$. In the sequel we show a more sophisticated way of selecting the primitive feedback polynomial and seed of the LFSR-based PG which finds shorter sequences in less or competitive time. Given a specific feedback polynomial, we guide the search for the optimal seed by using the theory of *discrete logarithms*.

When initialized to a non-zero state, a n-stage LFSR with a primitive feedback polynomial cycles through all $2^n - 1$ non-zero binary n-tuples. When the feedback polynomial is changed from one primitive polynomial to another the order in which the patterns appear changes. Hence, a n-stage LFSR with a primitive feedback polynomial defines a *permutation* over the non-zero binary n-tuples, each polynomial corresponding to a different permutation.

Given a subset of binary n-tuples, the minimum subsequence of a LFSR needed to cover all the tuples of the subset will vary depending on the permutation defined by the feedback polynomial. If the position of the tuples in the permutation was known, it would be straight forward to find the minimum covering subsequence. The position of a tuple in a sequence can be obtained from the theory of *discrete logarithms.*

Let $f(x) = \sum_{i=0}^{n} f_i x^i = x^n + h(x)$ be a primitive polynomial of degree n over $GF[2]$ and let α be a root of f. The non-zero elements of $GF[2^n]$ can all be expressed as distinct powers of α, i.e.

$$\forall \beta \neq 0 \in GF[2^n],\ \exists\ 0 \leq j \leq 2^n - 2\ \ s.t.\ \beta = \alpha^j \tag{2.8}$$

Since α is a root of f we have $\alpha^n = h(\alpha)$, hence β can be represented as a unique non-zero polynomial in α of degree less than n.

If β is the j-th power of α, j is said to be the *discrete logarithm* of β to the base α. The discrete logarithm problem can be stated as follows: given α (equivalently, given f) and β, find j.

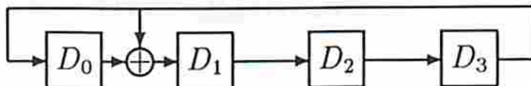

Figure 2.1: A LFSR with the feedback polynomial $f(x) = x^4 + x + 1$.

Discrete logarithms relate to the order in which patterns are generated by a LFSR as follows. Consider a LFSR with f as its feedback polynomial, as in Figure 2.1. We number the cells $D_0, D_1, \ldots, D_{n-1}$, with the feedback value coming out of cell D_{n-1} and feeding D_i iff $f_i = 1$. Consider the following mapping between non-zero patterns of the register and non-zero polynomials in α of degree less than n. A "1" in cell D_i represents α^i. Since every non-zero element of $GF[2^n]$ corresponds to a unique polynomial in α of degree less than n, the non-zero elements of $GF[2^n]$ correspond to the patterns of the register. Assume the initial state has a "1" in the least significant cell and 0 in all other cells. A shift will move the "1" to the second cell, corresponding to multiplication by α. After the n-th shift, the "1" is fed back, corresponding to substituting $h(\alpha)$ for α^n. Thus, the j-th pattern, interpreted as a polynomial in α, represents the element α^j. If we want to know when a pattern will appear we need to find the discrete logarithm of the element corresponding to the pattern.

We considered two algorithms for solving the discrete logarithm problem. The first is due to Pohlig and Hellman [32] and the second is due to Coppersmith [9]. An excellent expository of these two algorithms and of the discrete logarithm problem can be found in [28].

To avoid a lengthy discussion on the exact analysis of the computational complexity of these two algorithms we refer the reader to the above references. We just state the issues which influenced our decision to prefer the Pohlig-Hellman algorithm over Coppersmith's. While Coppersmith's algorithm has an asymptotic run time which is better than the Pohlig-Hellman algorithm, it is also much more complex. The computational complexity of the Pohlig-Hellman algorithm is dependent on the size and multiplicity of the prime factors of $2^n - 1$ (the number of non-zero elements in the field of computation). When the number of circuit inputs is less than 64, which

is true for the cases we are targeting, the prime factors of $2^n - 1$ are small enough (except for $n =$ 62,61,59, 49,41,37,31) for the Pohlig-Hellman algorithm to run faster than Coppersmith's.

2.5 The test embedding problem

In this section we define and present an algorithm to solve the *test embedding problem.* We analyze the computational complexity of our algorithm and discuss a strong limitation. We then present a way to overcome this limitation.

The *test embedding problem* is defined as follows. Given (1) a set of faults $F = \{f_1, \ldots, f_s\}$, (2) a set of test patterns $T = \cup_{i=1}^{s} T_i$, with $T_i = \{t_{i,1}, \ldots, t_{i,n_i}\}$ being the set of *all* tests for fault f_i, and $t_{i,j}$ being a binary n-tuple, and (3) a primitive polynomial, p, of degree n, find a minimum length subsequence of patterns generated by the corresponding LFSR that includes at least one test pattern for each f_i in F.

When given a set $P = \{p_1(x), \ldots, p_u(x)\}$ of primitive polynomials we would like to select the polynomial that generates the shortest such subsequence.

Having found the polynomial, the initial state of the LFSR and the sequence length, we have *embedded* a test set for F in the sequence that will be generated by the LFSR. The resulting test sequence is referred to as the *embedded (test) sequence.* The faults in F are referred to as *embedded faults* and the test patterns of T are the *embedded test patterns.*

To solve this problem we propose a two-stage procedure, referred to as the *embedding procedure (EP).* The first stage is referred to as the *discrete logarithm stage* and the second stage is referred to as the *windowing stage.*

In the *discrete logarithm stage* we first find the logarithm of all the test patterns $t_{i,j}$ with respect to a root α of the primitive polynomial p. We then sort the logarithms and for each logarithm we create a list of faults detected by the corresponding pattern.

In the *windowing stage* the idea is to use a sliding window on the cycle of logarithms, identifying those windows that detect all the faults, and selecting the smallest window (or one of the windows of shortest length when there are more than one). The outline of the procedure for this stage is given in Figure 2.2.

The array *covered*[] keeps track of the number of patterns in the window that detect each fault. The counter *not_covered* keeps count of the number of faults that are not detected by the patterns in the window. The array *log_table*[] stores the ordered logarithms, while the array *pattern_table*[] stores the patterns corresponding to the ordered logarithms. The procedure begins with the window containing the first logarithm. It is then extended, until all the faults are detected. The size of the window is recorded. It is then compared with the previous best and the smaller of the two is kept. At the end of each iteration the *tail* is advanced. In the following iteration the *head* is adjusted, if necessary, so that the window detects all the faults. The procedure terminates when all the logarithms have been considered as *tails*.

Denote the ordered logarithm cycle by $lg_1, lg_2, \ldots, lg_{|T|}$ where lg_i is followed by lg_{i+1} and $lg_{|T|}$ is followed by lg_1. Denote the window whose *tail* is lg_i by w_i and denote the *head* of w_i by h_i. We have the following Lemma.

Lemma 1: The head of w_{i+1}, h_{i+1}, is in the subcycle $[h_i \ldots lg_{i+1})$, i.e. it is not in the subcycle $[lg_{i+1} \ldots h_i)$.

Proof: Assume h_{i+1} is in the subcycle $[lg_{i+1} \ldots h_i)$, then the window $[lg_i \ldots h_{i+1}]$ detects all faults, contradicting the minimality of w_i. □

As a result of Lemma 1, Procedure *window*() finds the shortest window associated with each *tail*. Since the procedure considers windows beginning at all the logarithms and considers the logarithms of all the test patterns for all the faults, the procedure finds a smallest subsequence that detects all faults.

The complexity of Procedure *window*() is a function of the number of *tail* and *head* movements taken. There are at most $|T|$ logarithms, hence at most $|T|$ *tail* movements. Each window contains at most $|T|$ logarithms, hence there are at most $2|T|$ *head* movements. For each *tail* or *head* movement there are at most $|F|$ accounting operations that are needed, hence the complexity is $O(|T||F|)$.

Procedure 1: $window(n, \#_of_flts, \#_of_logs, log_table, pattern_table)$

1. $best = 2^n - 1$
2. for $(i = 0; i < \#_of_flts; i++)$
   (a) $covered[i] = 0$
3. $not_covered = \#_of_flts$
4. $end = \#_of_logs - 1$
5. for $(start = 0; start < \#_of_logs; start++)$
   (a) $tail = log_table[start]$
   (b) while$(not_covered > 0)$
      i. $head = log_table[(++end) \pmod{\#_of_logs}]$
      ii. for all faults j covered by $head$
         A. if$(covered[j] == 0)$, $not_covered--$
         B. $covered[j]++$
   (c) $size = head - tail + 1 \pmod{2^n - 1}$
   (d) if $best > size$
      i. $best = size$
      ii. $seed = pattern_table[start]$
   (e) for all faults j covered by $tail$
      i. if$(covered[j] == 1)$, $not_covered++$
      ii. $covered[j]--$

Figure 2.2: The procedure for the *windowing stage* of *EP*

The complexity of *EP* is given by $O(PPDL + |T|(DL + \log|T| + |F|))$, where $PPDL$ is the preprocessing effort required for the Pohlig-Hellman algorithm, DL is the time required for one logarithm computation and $|T|\log|T|$ is the time required to sort the logarithms.

As mentioned in the previous section, for the cases we are targeting $PPDL$ and DL will not require much effort. The bulk of the work required by $PPDL$ is the construction of a hash table of size $2 \cdot \sum_{i=1}^{m} q_i$, where m is the number of distinct prime factors of $2^n - 1$ and the q_i's are the distinct prime factors. Each entry involves one polynomial multiplication and one polynomial reduction modulo p and one insert operation into the table. The work required for one DL operation is $O(mn)$ polynomial multiplications and reductions modulo p and m integer multiplications and reductions.

The major factor in the complexity expression is the number of test patterns, $|T|$, which might be overwhelming, rendering the algorithm impractical. We must, therefore, find a way to limit the size of T. This is done in two ways, based on a *user-set limit* on the effort allowed for embedding a fault. The first is by considering only a subset of all possible faults, those which will be classified as *hard*. For some circuits the set of hard faults will be empty. These circuits are classified as *randomly testable* circuits and no embedding is done for them. A short test sequence for these circuits is found by random selections. The second is by limiting for each *hard* fault the number of test patterns that will participate in *EP* (whose logarithms we compute). This limit will typically apply to only a small subset (if at all) of the hard faults. In the next section we describe our heuristic for identifying hard faults.

2.6 Identifying hard faults

The amount of work needed for *EP* is strongly affected by the number of test patterns. We have to modify the procedure in a way that will reduce the test set to a manageable size. The modification is based on partitioning the set of faults in a circuit into two sets - *randomly testable faults* and *random resistant faults*, with each set being possibly empty. By *randomly testable faults* we mean faults that can be

detected by choosing a primitive feedback polynomial and an initial seed at random, and using a test sequence of *acceptable length* (this term will be defined later). By *random resistant faults* we mean faults that cause the test length (for 100% fault coverage) to dramatically increase beyond the *acceptable length* when the feedback polynomial and the initial seed are chosen at random.

This partition is justified by the super-exponential decrease in the probability that a fault is missed by a random test sequence as the number of test patterns for the fault is doubled.

Our thesis is that by proper identification of the *random resistant faults* (referred to as *hard faults* in the sequel), and by embedding test patterns for each of these faults in a LFSR subsequence, the subsequence will also detect all the *randomly testable faults.*

As was shown in Section 2.1, the probability that a random sequence does not detect a fault in C_k or C_{k+1} is (much) greater than the probability that the sequence does not detect any other fault, hence any procedure that identifies hard faults must be able to identify the faults in both these subsets. We refer to the union of C_k and C_{k+1} as HF.

Before presenting our procedure for identifying hard faults, we present a brief discussion on the applicability of our algorithm. Every circuit can be associated with parameters (n, k). These parameters determine the *work factor* for *EP* and the *predicted length* of the embedded test sequence. We expect the embedded sequence to be of length between $L/2 = 2^{n-k-1}$ and $L = 2^{n-k}$. When embedding test patterns we ensure that in the resulting test sequence the embedded faults will be detected and we rely on chance that the non-embedded faults will also be detected. It follows that we must embed the faults in HF. Assuming little overlap between the test sets for each of the faults, the embedded test set is at least of order $|T_{emb}| = 2^k \cdot |C_k| + 2^{k+1} \cdot |C_{k+1}|$. The size of T_{emb} increases with k, whereas the predicted test length decreases. The applicability of our algorithm depends on the amount of preprocessing work (embedding) we are willing to do and the amount of time we are willing to allow the test session (test length).

The bound we place on the amount of pre-processing work determines what we consider as a *hard* circuit and what we consider as an *easy* circuit. If we allow each embedded fault at most 2^{δ} test patterns, then for a circuit for which $k = \delta$ we expect to find an embedded sequence of length longer than $2^{n-\delta-1}$. Thus, if for a given circuit we find a random test sequence of length less than $2^{n-\delta-1}$, we consider the circuit to be randomly testable, i.e. *easy*. In our experiments we chose to allow each embedded fault at most $2^{13} - 2^{14}$ test patterns. This led us to classify circuits as *easy* when we found a random test length that was less than 2^{n-15} (the *acceptable length*).

The bound we impose on the test length determines whether or not the resulting embedded sequence is applicable to a circuit. For example, if for a 32 input circuit we find that the hardest faults have 2^6 test patterns, we expect to find a test sequence of length $2^{25} - 2^{26}$. If the maximum allowed test length is 2^{20}, then our scheme is not practical for this circuit. On the other hand, no other scheme that uses just one seed and no reconfiguration of the LFSR will detect 100% of the faults within the imposed time constraints. We denote the maximum allowed test length by 2^{max}.

In the remainder of this section we present our heuristic for identifying hard faults followed by a probabilistic analysis of its effectiveness. The experimental results validate the proposed process.

2.6.1 The heuristic

The process by which we identify hard faults (referred to as *IHF*) is made of three phases. In the first phase we determine which of the faults of interest are redundant. They are eliminated from the set. The second and third phases are based on simulation (sampling) experiments. The goal of the second phase is to determine an estimate l for k. The goal of the third phase is to identify the random resistant faults using our estimate for k.

The second phase should be a fast process that comes as close as possible to identifying k. The outline of this phase is given in Figure 2.3. The procedure determines a cutoff value, *undet*, which we set to be the minimum between 50

Procedure 2: *second_phase*$(circuit, k, max, \#irredundant_flts)$

1. $undet = \min\{50, 0.05 \cdot \#irredundant_flts\}$
2. for($j = n - 15$; $j \leq max$; $j++$)
   (a) $ND = \{$ faults not detected by a random sequence of length 2^j $\}$
   (b) if(($j = n - 15$) and ($ND = \phi$)) return(-2) /* the circuit is *easy* */
   (c) if $|ND| < undet$, break
3. if ($j > max$) return(-1) /* the expected sequence length is unacceptable */
4. $UND = ND$
5. for($i = 0$; $i < 4$; $i++$)
   (a) $ND = \{$ faults not detected by a random sequence of length 2^j $\}$
   (b) $UND = UND \cup ND$
6. $t = \min_i\{|T_i|\}_{i=1}^{|UND|}$ where T_i is the set of all test patterns for the i-th fault in UND.
7. return($\lceil \log t \rceil$)

Figure 2.3: The procedure for the second phase of identifying hard faults

and 5% of the irredundant faults. It applies random test sequences on the circuit, with increasing lengths, until a sequence that misses at most *undet* faults is found. The initial sequence length is $2^j = 2^{n-15}$. It is understood that $(n - 15) \leq max$, otherwise the procedure will not result in an embedded test sequence we are willing to use, hence there is no sense in carrying it out. If during this iteration a sequence is generated that detects all faults then a satisfying embedding for *all* the faults is found and the circuit is considered *easy*. Otherwise, j is incremented and another fault simulation is performed. This process is repeated (each time simulating all faults) until the number of undetected faults is less than *undet* or until j is greater than max. If $j > max$ the procedure stops, it will not produce a satisfying test sequence. Otherwise four more fault simulation experiments of length 2^j are conducted. For each of the five experiments the set of undetected faults is recorded and at the end of the experiments the union of these sets is constructed. For each fault in the union all test cubes are generated in order to find the fault with the fewest number of test patterns. [1] If the number of test patterns for this fault is t, then the estimate of k is $l = \lceil \log t \rceil$.

Remark 1: Although test generation is considered a hard problem, and, a fortiori, generation of all test patterns, this proved to be a very low cost task in our experiments, where the faults of interest were single stuck-at faults.

After the execution of Procedure *second_phase()*, a circuit is classified as either *easy* or *hard*. It is classified as *easy* if one of two conditions is met: (1) a random test sequence of length less than 2^{n-15} was found that detects all faults; or (2) the estimate l is greater or equal to 15. In all other instances a circuit is classified as *hard*.

For circuits classified as *hard*, we turn to the third phase. We run a set of fault simulation experiments (we ran 20) with $L/2 = 2^{n-l-1}$ test patterns. *The set of faults which are not detected by at least one experiment constitute the set of embedded faults (EF).* These are the faults for which we embed test patterns. It is

[1] A test pattern is a binary vector, indicating exact values in every position, whereas a test cube is a ternary vector, the third symbol being a *don't care* symbol.

essential that the faults in EF include all the faults of HF. The parameters used in selecting the number of simulation runs in phases two and three were chosen to ensure that the probability that EF includes HF is very close to 1 (this is shown in the next section). If a fault f is included in EF, we say it is *classified as hard.* Otherwise, we say it is *classified as easy.*

Remark 2: We conduct 5 fault simulation experiments in phase two in order to increase the probability that at least one fault of C_k will be in the union of the undetected faults. It is important that l be as close as possible to k because this will reduce the number of faults we have in EF (the smaller l is, the longer the simulation lengths in phase three are, decreasing the probability of not detecting a random fault, hence fewer faults are in EF), thus reducing the number of test patterns we need to embed. While this also reduces the probability of detection for some of the faults (as will be seen in the next section), it is a tradeoff that must be made.

In the sequel, whenever *EP* is mentioned, it is understood that *IHF* was used in a preprocessing step (stage 0 if you will) to create a reduced target fault set.

2.6.2 Probabilistic analysis of the heuristic

The purpose of *IHF* is twofold. First, to find all the faults in HF. Second, by our thesis, embedding test patterns for these hard faults results in a test sequence that detects all the faults of interest. We must answer two questions regarding our heuristic.

1. What is the probability that a fault that *should* be classified as *hard* (i.e. has less than 2^{k+1} test patterns) is classified as *easy*?
2. What is the probability that an *easy* fault will not be detected by the embedded sequence?

In answering these questions we assume that the embedded sequence is made of a totally random set of patterns, although, given that petterns are generated by a LFSR, this might not be completely accurate.

In answering the first question we compute the probability that a fault with t test patterns will not be classified as *hard* by *IHF*. We will be interested in the probability value when t is less than or equal to 2^{l+1}, where l is the *IHF* estimate for k.

Let $p_{miss}(2^{n-l-1}, t)$ be the probability that a test sequence of length 2^{n-l-1} does not detect a fault with t test patterns. This would require that the first pattern not detect the fault, the second not detect the fault, and so on. Assuming the patterns are completely random, uncorrelated, with selections done without replacement and the all zero pattern does not participate in the drawing of patterns (and is not one of the test patterns), we have

$$\begin{aligned} p_{miss}(2^{n-l-1}, t) &= \frac{(2^n-1)-t}{(2^n-1)} \cdot \frac{(2^n-2)-t}{(2^n-2)} \cdot \ldots \cdot \frac{(2^n-2^{n-l-1})-t}{(2^n-2^{n-l-1})} \\ &= \left(1-\frac{t}{(2^n-1)}\right) \cdot \left(1-\frac{t}{(2^n-2)}\right) \cdot \ldots \cdot \left(1-\frac{t}{(2^n-2^{n-l-1})}\right). \end{aligned}$$

Let $p_{detect}(2^{n-l-1}, t)$ denote the probability that a sequence of length 2^{n-l-1} detects a fault with t test patterns, then

$$p_{detect}(2^{n-l-1}, t) = 1 - p_{miss}(2^{n-l-1}, t).$$

Let $\#sim$ denote the number of simulation experiments. Then $p_{all_det}(2^{n-l-1}, t)$, the probability that a fault with t test patterns is detected by all the experiments is

$$p_{all_det}(2^{n-l-1}, t) = (p_{detect}(2^{n-l-1}, t))^{\#sim}.$$

$p_{all_det}(2^{n-l-1}, t)$ is also the probability that a fault with t test patterns will not be classified as hard. This probability can be increased or decreased by changing the number of simulation experiments. In our experiments, $p_{all_det}(2^{n-l-1}, 2^l)$ ranged

from $7.9 \cdot 10^{-9}$ to 10^{-8}. $p_{all_det}(2^{n-l-1}, 2^{l+1})$ ranged from 10^{-4} to $1.25 \cdot 10^{-4}$. This suggests that with very high probability EF will include all the faults in HF. EF will typically include some other faults, those with higher detection probabilities than the faults in HF, thus although EF might vary from one run of EP to another, all sets will share the same core of "hardest" faults.

We turn to the second question, namely what is the probability, $p_{nd}(el, t)$, that an embedded sequence, es, of length el does not detect a fault f, where there are t possible test patterns for f. We first answer the complement question, that is, what is the probability that es detects f.

$$\begin{aligned} prob(es \text{ detects } f) &= prob(es \text{ detects } f \mid f \text{ is classified } hard) \cdot \\ &\quad prob(f \text{ is classified } hard) + \\ &\quad prob(es \text{ detects } f \mid f \text{ is classified } easy) \cdot \\ &\quad prob(f \text{ is classified } easy). \end{aligned}$$

By construction, es contains at least one test patterns for each of the hard faults, hence

$$prob(es \text{ detects } f \mid f \text{ is classified } hard) = 1.$$

By definition

$$prob(f \text{ is classified } hard) = 1 - p_{all_det}(2^{n-l-1}, t).$$

The probability $prob(es \text{ detects } f \mid f \text{ is classified } easy)$ is the probability that a random sequence of length el detects f. By definition

$$prob(f \text{ is classified } easy) = p_{all_det}(2^{n-l-1}, t).$$

Thus,

$$prob(es \text{ detects } f) = 1 \cdot (1 - p_{all_det}(2^{n-l-1}, t)) + p_{detect}(el, t) \cdot p_{all_det}(2^{n-l-1}, t)$$

$$= 1 - p_{all_det}(2^{n-l-1}, t) + p_{all_det}(2^{n-l-1}, t) -$$

$$p_{miss}(el, t) \cdot p_{all_det}(2^{n-l-1}, t)$$

$$= 1 - p_{miss}(el, t) \cdot p_{all_det}(2^{n-l-1}, t).$$

Hence

$$p_{nd}(el, t) = p_{miss}(el, t) * p_{all_det}(2^{n-l-1}, t).$$

From the above expression the probability of f not being detected is the probability that f is classified as *easy* (p_{all_det}) and a random sequence of length el fails to detect f (p_{miss}).

The probability $p_{nd}(el, t)$ is a product of two functions of t. But whereas one of these functions (p_{all_det}) increases with t, the other (p_{miss}) decreases. We focus our attention on the behavior of p_{nd} as a function of t. As will be seen in Theorem 2, p_{nd} has a maximum value and no local maxima, i.e. as t increases, p_{nd} rises, reaches a maximum value and decreases from there on. The overall probability that the embedded sequence detects all the faults is dependent upon the probability of occurrence of the faults whose number of test patterns is in the peak area. The values of $p_{nd}(el, t)$ for $t = 2^j$, $l \leq j \leq l + 5$, from our experiments are tabulated in Table 2.10. The peaks can clearly be seen.

Let $N = 2^n$, $r = 2^{n-l-1}$, $q = el$ and $s = \#sim$. We can rewrite p_{nd} as follows

$$p_{nd}(N, q, t, r, s) = \prod_{v=N}^{N-q+1} \left(\frac{v-t}{v}\right) \left[1 - \prod_{u=N}^{N-r+1} \left(\frac{u-t}{u}\right)\right]^s$$

We assume $t \leq N - q$ since if $t > N - q$, then a test sequence of length q will always have to include at least one of the t test patterns, hence p_{nd} will be zero.

Theorem 2 shows that once p_{nd}, as an integer function in t, either levels off or starts falling, it will keep on falling, hence it has one maximum value with no local maxima.

Theorem 2: If $p_{nd}(N, q, t, r, s) \geq p_{nd}(N, q, t + 1, r, s)$ then

1. For all i s.t. $t+1 \leq i < \min\{N-q, N-r\}$

$$p_{nd}(N,q,i,r,s) > p_{nd}(N,q,i+1,r,s)$$

2. If $N-r < N-q$, then for all i s.t. $t+1 < N-r \leq i < N-q$

$$p_{nd}(N,q,i,r,s) > p_{nd}(N,q,i+1,r,s)$$

□

Proof: We look at the ratio $p_{nd}(N,q,t,r,s)/p_{nd}(N,q,t+1,r,s)$ and we show that if this ratio is greater or equal to 1, then all succeeding ratios will be greater than 1. We begin with the first part of the theorem.

$$\frac{p_{nd}(N,q,i,r,s)}{p_{nd}(N,q,i+1,r,s)} = \frac{\prod_{v=N}^{N-q+1}\left(\frac{v-i}{v}\right)\left[1-\prod_{u=N}^{N-r+1}\left(\frac{u-i}{u}\right)\right]^s}{\prod_{v=N}^{N-q+1}\left(\frac{v-(i+1)}{v}\right)\left[1-\prod_{u=N}^{N-r+1}\left(\frac{u-(i+1)}{u}\right)\right]^s}$$

Denote

$$U_i = \prod_{u=N}^{N-r+1}\left(\frac{u-i}{u}\right)$$

then

$$\frac{p_{nd}(N,q,i,r,s)}{p_{nd}(N,q,i+1,r,s)} = \frac{N-i}{N-i-q}\left[\frac{1-U_i}{1-U_{i+1}}\right]^s.$$

By the hypothesis, for $i=t$, the ratio is greater or equal to 1, hence

$$\begin{aligned}\left[\frac{1-U_i}{1-U_{i+1}}\right]^s &\geq \frac{(N-i-q)}{(N-i)} \\ &= 1-\frac{q}{N-i}.\end{aligned}$$

Denote the above inequality as

$$M_i^s \geq Q_i.$$

For $i \geq t$, $Q_i > Q_{i+1}$. If we show that $M_{i+1} > M_i$ then we get

$$M_{i+1}^s > M_i^s \geq Q_i > Q_{i+1}$$

which proves the first statement. To show that $M_{i+1} > M_i$ we need to show that

$$(1 - U_{i+1})^2 - (1 - U_i)(1 - U_{i+2}) > 0. \tag{2.9}$$

We use the fact that

$$\frac{U_i}{U_{i+1}} = \frac{N-i}{N-i-r} = 1 + \frac{r}{N-i-r}$$

and that

$$\frac{U_{i+2}}{U_{i+1}} = \frac{N-i-r-1}{N-i-1} = 1 - \frac{r}{N-i-1}$$

to show

$$\begin{aligned} (1 - U_{i+1})^2 - (1 - U_i)(1 - U_{i+2}) &= \frac{r(r-1)}{(N-i-1)(N-i-r)} U_{i+1} + \\ &\quad \frac{r}{(N-i-1)(N-i-r)} U_{i+1}^2 \\ &> 0. \end{aligned}$$

The last inequality in based on the fact that $i < min(N-q, N-r)$.

To prove the second statement, notice that for $i = N - r$, U_{i+1} and U_{i+2} are both equal to zero, hence Equation (2.9) becomes $U_i > 0$ and there is nothing to prove. For $i > N - r$, a test sequence of length r will always include at least one of the i test patterns, hence p_{all_det} is equal to 1. The ratio becomes

$$\frac{\prod_{v=N}^{N-q+1} \left(\frac{v-i}{v}\right)}{\prod_{v=N}^{N-q+1} \left(\frac{v-i-1}{v}\right)} = \frac{N-i}{N-q-i} > 1$$

□

2.7 Experimental results

Experiments were conducted using a modified versions of the ATALANTA fault simulation and test generation system [18]. These modifications included fault simulating random patterns generated by LFSRs with primitive feedback polynomials, and generating all test cubes per fault (or as many as ATALANTA could find, depending on the allowed backtrack limit) instead of stopping once the first test pattern is found.

We first tried *EP* on the ISCAS85 benchmark circuits. The characteristics of some of these circuits, as well as the results of the experiments are given in Table 2.4. The first five columns of the table are self explanatory. The sixth column (#nr.flts.) gives the number of irredundant faults for each circuit. All of these circuit were classified as *easy* by Procedure *second_phase*(). Column seven (sp_time) is the CPU time (in seconds) required for Procedure *second_phase()* on a SPARC 2 workstation.

Since these circuits were classified as *easy*, we did not continue on with *EP*. To get an idea of the required pseudo-random test length for these circuits, we conducted 10 *random selection* experiments (referred to as *RS* experiments in the sequel) for each circuit. In each RS experiment a primitive feedback polynomial and an initial seed were chosen at random and test patterns generated until 100% fault coverage was achieved. Columns eight and nine give the length of the best (shortest) and worst (longest) sequence that was required for each of the circuits. By looking at the values in column ten, we see that the sequence lengths that were found are much smaller than 2^{n-15}.

We did not try to embed test sequences for circuits $c2670\,(n = 233)$, $c5315\,(n = 178)$, and $c7552\,(n = 207)$ because the number of inputs for each of these circuits made the embedding impractical. These circuits are either randomly testable or the embedded test length would be so long (greater than 2^{n-15}) that test set embedding would not be a viable test option.

Table 2.4: Experimental results for some ISCAS85 combinational benchmarks.

circuit	#pi	#po	#gates	depth	#nr.flts	sp_time	best	worst	$\#pi - 15$
c432	36	7	196	18	520	1	480	3136	21
c499	41	32	243	12	750	1	896	1536	21
c880	60	26	443	25	942	1	7520	34176	45
c1355	41	32	587	25	1566	2	1344	2848	26
c1908	33	25	913	41	1870	4	5216	33152	18
c3540	50	22	1719	48	3291	3	8960	50016	35
c6288	32	32	2448	125	7710	13	64	256	17

Since the ISCAS circuits do not provide a good test bed for our algorithm, we turned to the Berkeley circuits [6]. We synthesized circuits with 20 to 40 inputs as multilevel circuit. Some circuits were eliminated for pathological reasons.

The characteristics of the synthesized circuits are given in columns two (#pi) to six (#nr.flts) of Table 2.5. Column eight (sp_time) is the CPU time, in seconds, required to run the second phase on a SPARC 2 workstation. The second and third phase of IHF were combined for circuit *xparc*. The reason for this was the long simulation time required for this circuit. Thus, no time is reported for this circuit in Table 2.5.

To identify the randomly testable circuits from the random resistant ones we ran each circuit through *IHF*. The simulation experiments of the second and third phase of *IHF* used a pool of 150 primitive polynomials. For a circuit with n inputs we used the first 150 primitive polynomials of degree n, when ordered lexicographically (i.e. $(x^n + x + 1) > (x^n + 1)$).

The results of Procedure *second_phase()* on these circuits is given in column seven (l) of Table 2.5. For each of the *easy* circuits we ran 10 RS experiments to find a short pseudo-random test sequence. We also conducted a more thorough search for k for these circuits. We allowed each RS experiment to generate at most 2^{n-l+1} test patterns, where l is the estimate for k. The results of these experiments are in Table 2.6. The value of $n - l$ is given in column two. The best (shortest sequence) and worst (longest sequence) results are given in columns three and four. None of the worst sequences achieved 100% fault coverage. The total CPU time needed for

Table 2.5: Characteristics of synthesized circuits.

circuit	#pi	#po	#gates	depth	#nr.flts	l	sp_time
b4	33	22	171	9	562	*easy*	2
in6	33	23	172	9	564	*easy*	2
jbp	36	57	353	13	1051	*easy*	3
misj	35	12	63	5	89	*easy*	1
signet	39	8	243	8	709	*easy*	1
x6dn	38	5	207	13	651	*easy*	2
bc0	21	11	372	15	1434	4	21
b3	32	19	415	17	1283	12	63
chkn	29	7	282	13	959	5	997
cps	24	109	486	18	1837	3	890
exep	28	62	256	12	995	11	21
in3	34	29	240	14	714	14	77
in4	32	20	422	23	1303	12	70
in5	24	14	192	14	630	10	6
in7	26	10	117	12	330	9	5
vg2	25	8	189	12	576	7	8
vtx1	27	6	207	12	616	7	72
xparc	39	69	729	21	2614	12	*
x1dn	27	6	207	12	616	7	34
x9dn	27	7	215	12	636	7	83

Table 2.6: RS results for the randomly testable synthesized circuits.

circuit	$n-l$	best	worst	$n-15$
b4	15	48,244	130,016	18
in6	15	55,712	130,016	18
jbp	13	20,064	32,768	21
misj	10	672	4096	20
signet	13	7,232	32,768	24
x6dn	14	22,368	65,536	23

these experiments was $1-2$ minutes, as reported by ATALANTA on a SPARC 1 workstation. The best random sequence that was found was always shorter than 2^{n-l+1} ($2L$). Also, $n-l+1$ was always less than $n-15$ (column five).

For the circuits classified as *random resistant circuits* we tried to find short test sequences in two ways. The first was by RS experiments and the second was by continuing our embedding procedure (the first two phases of *IHF* were already executed).

The purpose of finding a minimum sequence in two ways was to evaluate the quality of the result of *EP* in terms of test length and processing time.

We ran 100 RS experiments for each circuit, except for circuit *chkn* for which we ran just 50 experiments and circuit *xparc* for which we ran just 3 experiments. The results of these experiments are in Table 2.7. Column *rs_time* represents the total CPU time (read as $hrs : min$) needed for these experiments, as reported by ATALANTA on a SPARC 2 workstation. We allowed each RS experiment to run for at most 2^{n-l+1} test patterns. For all the circuits, the worst sequence lengths are the maximum allowed lengths per experiment and all such sequences did not detect all the faults. Comparing the best sequence length and the value 2^{n-l}, we see that the best value was less than 2^{n-l} (L) for 8 of the circuits, while it was between L and $2L$ for the remaining 6 circuits.

For the random resistant circuits we completed the execution of *IHF* and proceeded to embed the test patterns for these faults. In Table 2.8, the results of *IHF* are shown. For each circuit, we show the total number of irredundant faults, the

Table 2.7: RS results for the random resistant synthesized circuits.

circuit	#sim.	best	2^{n-l}	worst	rs_time
bc0	100	114,624	131,072	262,144	0:36
b3	100	487,168	1,048,576	2,097,152	6:00
chkn	50	15,561,920	16,777,216	20,000,000	25:15
cps	100	2,292,512	2,097,152	4,194,304	11:46
exep	100	155,584	131,072	2621448	0:32
in3	100	660,224	1,048,576	2,097,152	4:29
in4	100	548,480	1,048,576	2,097,152	5:19
in5	100	11,424	16,384	32,768	0:04
in7	100	56,544	131,072	262,144	0:25
vg2	100	267,552	262,144	524,288	1:01
vtx1	100	1,345,152	1,048,576	2,097,152	4:29
xparc	3	248,802,208	134,217,728	268,435,456	41:36
x1dn	100	1,948,192	1,048,576	2,097,152	4:30
x9dn	100	1,022,944	1,048,576	2,097,152	4:49

number of faults that were classified as hard and the percentage of hard faults. The total number of faults that were classified as hard, over all circuits, make up 7% of the total number of faults in the circuits. Thus, when also considering that the hard faults are those with the fewest test patterns, several orders of magnitude of reduction in the work effort is achieved when embedding only the hard faults.

In Table 2.9 we show the results of *EP* for each of the random resistant circuits. The second column shows the total number of embedded patterns, i.e. the union of the sets of test patterns for the hard faults. Column three shows the number of different primitive polynomials considered, i.e. the minimum length embedded sequence was computed for each of these polynomials. The polynomials we used were the first in the lexicographical ordering. Columns four and five give the best and worst embedding results for these polynomials, and column six gives the total CPU time (read as $hrs : min$) required for *EP*, using an HP-700 workstation. When comparing the best sequence length with the value 2^{n-l} for each of the circuits (see Table 2.7), we see that the length of the best embedded sequence was always less than 2^{n-l} except for the circuit *exep*. For six of the circuits, the worst sequence length was also less than 2^{n-l}. When comparing the worst embedding length with

Table 2.8: Hard fault identification for synthesized circuits.

circuit	#nr.flts.	#h.flts.	%	circuit	#nr.flts.	#h.flts.	%
bc0	1434	65	4.53	b3	1283	29	2.26
chkn	959	88	9.18	cps	1837	388	21.12
exep	995	86	8.64	in3	714	16	2.24
in4	1303	28	2.15	in5	630	77	12.22
in7	330	8	2.42	vg2	576	15	2.60
vtx1	616	29	4.71	xparc	2614	130	4.97
x1dn	616	28	4.55	x9dn	636	43	6.76

the best RS length, for 6 of the circuits the worst embedding length was shorter than the best RS length.

From the embedding length for each circuit we can calculate the probability $p_{nd}(el, t)$ that an embedded sequence of length el will not detect a fault that has t test patterns, for various values of t. This is shown in Table 2.10. The values in the parenthesis in each table entry are exponents of 10. For example, the value of $p_{nd}(el, 2^l)$ for circuit *bc0* is $5.67 * 10^{-9}$. Depending on the values, one might decide that the probability of not detecting a fault is small enough such that verification by simulation is not necessary.

Having performed the RS experiments and the embedding experiments for the random resistant circuits, we proceeded to compare the results in terms of the shortest sequence length found and the processing time for the experiments. These comparisons are shown in Table 2.11. Columns two and three give the best test length and the time required for the RS experiments. Columns four and five give these values for the embedding experiments. In column six we give the ratio between the best RS sequence length and the best embedding sequence length. Column seven gives the ratio between the processing time of the RS experiments and the embedding experiments. These ratios were *normalized*, so they would reflect the ratio if both experiments were run on a SPARC 2. To find the normalization factor we ran *EP* for circuit *chkn* on a SPARC 2 workstation and compared the run time with the run time for the same procedure and circuit on the HP-700. The normalization

Table 2.9: Embedding results for the synthesized circuits.

circuit	#emd.vecs.	#pols.	best	worst	emb_time
bc0	3,120	50	74,240	196,741	0:30
b3	171,707	4	385,824	614,368	7:01
chkn	13,328	20	9,459,968	12,551,540	1:02
cps	9,024	25	1,254,272	3,464,351	1:27
exep	281,235	2	149,120	150,912	4:22
in3	148,683	4	176,960	239,296	3:27
in4	194,556	3	530,752	617,600	5:51
in5	176,310	3	6,530	7,416	3:22
in7	9,728	15	9,274	29,488	0:24
vg2	2,556	100	111,456	320,438	0:35
vtx1	5,216	25	838,176	1,621,458	0:41
xparc	536,818	1	133,667,296	*	13:20
x1dn	5,216	25	838,176	1,621,458	0:41
x9dn	6,144	25	446,624	1,200,397	0:45

Table 2.10: $p_{nd}(el, t)$ values for $t = 2^j$, $l \leq j \leq l + 5$.

circuit	(n, l)	$(el, 2^l)$	$(el, 2^{l+1})$	$(el, 2^{l+2})$	$(el, 2^{l+3})$	$(el, 2^{l+4})$	$(el, 2^{l+5})$
bc0	(21,4)	5.67(-9)	3.90(-5)	5.99(-3)	7.01(-3)	*	*
b3	(32,12)	5.48(-9)	5.00(-5)	1.25(-2)	3.64(-2)	2.76(-3)	8.00(-6)
chkn	(29,5)	5.06(-9)	3.64(-5)	5.89(-3)	7.38(-3)	1.11(-4)	1.24(-8)
cps	(24,3)	6.96(-9)	4.33(-5)	5.53(-3)	5.00(-3)	4.77(-5)	2.30(-9)
exep	(28,11)	2.54(-9)	1.07(-5)	5.76(-4)	7.68(-5)	1.23(-8)	1.52(-16)
in3	(34,14)	6.68(-9)	7.40(-5)	2.78(-2)	1.79(-1)	6.67(-2)	4.51(-3)
in4	(32,12)	4.77(-9)	3.80(-5)	7.21(-3)	1.20(-2)	3.02(-4)	9.23(-8)
in5	(24,10)	5.33(-9)	4.69(-5)	1.11(-2)	2.85(-2)	1.68(-3)	2.85(-6)
in7	(26,9)	7.43(-9)	9.06(-5)	4.12(-2)	3.93(-1)	3.20(-1)	1.04(-1)
vg2	(25,7)	5.33(-9)	4.52(-5)	1.00(-2)	2.30(-2)	1.09(-3)	1.21(-6)
vtx1	(27,7)	3.66(-9)	2.10(-5)	2.24(-3)	1.13(-3)	3.00(-6)	7.18(-12)
xparc	(39,12)	2.93(-9)	1.42(-5)	1.02(-3)	2.39(-4)	1.19(-7)	1.44(-14)
x1dn	(27,7)	3.66(-9)	2.10(-5)	2.24(-3)	1.13(-3)	3.00(-6)	7.18(-12)
x9dn	(27,7)	5.32(-9)	4.50(-5)	1.00(-2)	2.28(-2)	1.08(-3)	1.18(-6)

Table 2.11: Comparison of embedding results and random selection result.

	random selection		embedding		rs/emb				
circuit	test length	time	test length	time	test length	time	l	$n-l$	$n-2l$
in5	11,424	0:04	6,530	3:22	1.75	0.01	10	14	4
exep	155,584	0:32	149,120	4:22	1.04	0.06	11	18	7
b3	487,168	6:00	385,824	7:01	1.26	0.43	12	20	8
in4	548,480	5:19	530,752	5:51	1.03	0.46	12	20	8
in7	56,544	0:25	9,274	0:24	6.10	0.52	9	17	8
bc0	114,624	0:36	74,240	0:30	1.54	0.60	4	17	13
in3	660,224	4:39	176,960	3:27	3.73	0.68	14	20	6
vg2	267,552	1:01	111,456	0:35	2.40	0.87	7	18	11
xparc	248,802,208	41:36	133,667,296	13:20	1.86	1.56	12	27	15
x9dn	1,022,944	4:49	446,624	0:45	2.29	3.21	7	20	13
vtx1	1,345,152	4:29	838,176	0:41	1.60	3.28	7	20	13
x1dn	1,948,192	4:30	838,176	0:41	2.32	3.29	7	20	13
cps	2,169,248	11:46	1,254,272	1:27	1.73	4.06	3	21	18
chkn	15,561,920	25:15	9,459,968	1:02	1.65	12.22	5	24	19

factor came out to be 2. Columns eight, nine and ten show the values of l, $n-l$, and $n-2l$ respectively.

The first fact we can notice from Table 2.11 is that the test length resulting from *EP* is always shorter than the test length resulting from the RS experiments. The second fact we notice is that the ratio of the processing times required by the two procedures varies. This can be explained as follows. The factor that affects the processing time for *EP* is the number of embedded patterns, which is strongly affected by the value of l. *EP* will require less time for circuits with lower values of l (omitting the obvious influence of the number of embedded faults and the number of polynomials). The factor that affects the processing time for the RS experiments is the length of each experiment, i.e. the value of $(n-l)$. The higher the value of l, the lower the value of $(n-l)$, resulting in lower processing time for the RS experiments. Notice that for the eight circuits whose l-value is less than 10, the time ratio is greater than 1 for five and greater than 0.5 for all eight. In absolute time, *EP* required an hour or less for seven of the eight and an hour and a half for the eighth. Of the eight circuits for which the value $n-2l$ was greater than 10, for six of the eight, *EP* required less time than the RS experiments. Define the product of the test length ratio and the processing time ratio (columns six and seven)

as a measure of EP effectiveness. When this measure is greater than 1 then any relative advantage RS has over EP in either length or time is negated by the bigger advantage EP has in the other. Except for circuit $bc0$ (whose measure was 0.924), the circuits with $n - 2l > 10$ all had effectiveness measures greater than 1. Of those, only $vg2$ had a time ratio less than 1. Circuits $in7$ and $in3$ also had effectiveness measures greater than 1. Altogether, 9 of the 14 circuits had effectiveness measures greater than 1.

Remark 3: The time reported for *EP* is the time required for the logarithm computations. It does not include the time required for phase three of *IHF*, the time it took to generate the test patterns and sort them before computing the logarithms, nor the time for Procedure $window()$. This is because these times were very small when compared with the time required by the RS experiments or the time required for logarithm computations. The time required for phase three was always between $5\% - 10\%$ of the RS time. For circuits *b3* and *in3*, generating and sorting the test patterns took just over $2:30$ minutes on a SPARC 2 and Procedure $window()$ took just over $4:00$ minutes for *b3* and less than that for *in3* on an HP-700. These times are insignificant when compared with the RS and logarithm computation times.

One can ask whether we really needed all the 100 RS experiments, or would 20 or 50 have been enough to produce a test sequence of length comparable with the one found by *EP*. A partial answer to this can be found in Table 2.12. The columns of Table 2.12 represent ratios between the sequence lengths from the RS experiments and the best embedding result. An integer entry in position (i, j) indicates the number of experiments for circuit i where the ratio was between the value of the headings for the j-th and $(j+1)$-th columns. For example, for circuit *cps* there was one RS experiments whose ratio was between 1.50 and 1.75. The last non-zero entry in each row includes all the experiments whose ratio was greater or equal to the ratio of the column it sits in, e.g. for circuit *in4* 70 experiments had a ratio greater than 2.50. These 70 experiments include all those that ran for the maximum allowed time and failed to detect all the faults.

Table 2.12: Distribution of simulation length vs. embedding length ratio.

	ratio						
circuit	1.00	1.25	1.50	1.75	2.00	2.25	2.50
bc0	*	*	1	1	2	3	93
b3	*	2	1	1	1	3	92
chkn	*	*	1	2	47	*	*
cps	*	*	1	0	4	4	91
exep	1	5	8	86	*	*	*
in3	*	*	*	*	*	*	100
in4	2	3	6	6	8	5	70
in5	*	*	1	3	1	3	92
in7	*	*	*	*	*	*	100
vg2	*	*	*	*	*	2	98
vtx1	*	*	3	0	2	1	94
xparc	*	*	*	1	2	*	*
x1dn	*	*	*	*	*	2	98
x9dn	*	*	*	*	*	2	98

The results in this table tend to support the notion that we didn't get "lucky" with random selections and usually all the experiments were necessary to ensure that we find a short sequence length.

For those circuits whose time ratio was less than one, we conducted additional RS experiments to bring the ratio up to one. The results of these experiments are in Table 2.13. For four of the eight circuits the additional experiments did not find shorter sequences. For two circuit, the additional experiments found shorter sequences, but still longer than the embedded sequence. For two circuits the additional experiments produced two sequences that had shorter lengths than the embedded sequences. Looking closer at these circuits, the embedding were done relative to only two and three polynomials and their length ratio relative to the first 100 RS experiments was poor to begin with. Thus, there was already indication that the polynomials used for the embeddings were poor choices.

Remark 4: As mentioned in Section 2.6, our thesis is that the test sequence found for the hard faults will also detect the easy faults. This was true in *all* the embedding experiments except for one polynomial for the circuit *in3*, in which the embedded

Table 2.13: Results of additional RS experiments to get equal time comparisons.

circuit	add. exp.	best result	best embedding
in5	9,900	8,192	6,530
exep	1,666	134,848	149,120
		144,864	
b3	132	611,296	385,824
in4	117	456,704	530,752
		501,536	
in7	92	68,064	9,274
bc0	66	163,680	74,240
in3	47	495,168	176,960
vg2	15	456,448	111,456

sequence missed nine irredundant faults, and three polynomials for circuit *in7*, of which two sequences missed one fault and one sequence missed eleven faults.

Remark 5: For circuits *in3, exep, b3, in4, in5, in7* and *xparc*, some of the hard faults had more than the maximum allowed number of test patterns (2^{14} for *in3*, 2^{12} for *exep* and 2^{13} for the others). For these faults we embedded only the maximum allowed number of tests. This caused the predicted length (the distance between the *head* and *tail* of the shortest window) for some of the embeddings to be greater than the actual length. This is a result of the fact that we did not embed *all* possible tests, but only a subset. Additional patterns were found in the embedded sequence, rendering some of the embedded patterns unnecessary, hence the shorter length. Thus, when embedding only subsets of test sets, the length of the embedded sequence is an upper bound on the actual test length. When the complete test sets are embedded the length of the embedded sequence is the actual test length.

2.8 Conclusions

In this chapter we address the question of finding *short* pseudo-random test sequences that achieve 100% fault coverage for LFSR-based PGs. We first show that if the probability of detecting the hardest fault in the circuit is p, then the probability

that a pseudo-random test sequence of length $\frac{2}{p}$ will achieve 100% fault coverage is typically greater than $\frac{1}{50}$. We then present an algorithm for embedding test patterns in test sequences generated by LFSRs. The algorithm is based on the theory of discrete logarithms. It produces a one-seed test sequence that detects all the faults of interest. The algorithm can also embed two-pattern tests and can also embed test patterns in sequences generated by CAs. The advantage of our approach over existing schemes that achieve 100% fault coverage is the low overhead we incur in terms of both circuit area and delay.

The applicability of the embedding algorithm depends on two user specified constraints. The first is the computational effort one is willing to expend (the parameter δ in Section 2.6.1) and the second is the length of the test session one is willing to allow (the parameter max in Section 2.6.1). Thus, for circuits where the hardest faults have more than 2^{δ} test patterns the computational effort required will be too high and for circuits with too many inputs, i.e. when $n - \delta - 1 > max$, the resulting sequence will be too long to be useful.

We find *short* test sequences either by the embedding algorithm or with random selections. This is decided by classifying circuits as either *randomly estable* or *random resistant*. This classification is based on the number of test patterns for the hardest fault in the circuit. If the logarithm of this number, denoted by k, is greater than δ, or if during the classification process a random sequence of length shorter than $2^{n-\delta-1}$ that achieves 100% fc is found, the circuit is classified as *randomly testable*. In all other cases it is classified as *random resistant*.

For *randomly testable* circuits our probabilistic and empirical results show that short test sequences can be found with few random selections.

For *random resistant* circuits we compared the results achieved by the embedding algorithm with those achieved by random selections of primitive polynomials and seeds for the LFSRs. In all cases the sequences found by our algorithm were shorter than those found by the random experiments. In almost half the cases, our algorithm was also faster than the random experiments. When the PG register is also to function as a RA, by considering only polynomials that achieve *zero-aliasing* (Chapter 3) as candidates for EP, both objectives are satisfied with one polynomial.

The circuits on which we ran the experiments are relatively small (only several hundred gates). Most of the effort for our algorithm is spent on logarithm computations, and only a small portion is spent on simulation. As circuit sizes increase, the cost of our algorithm will be only slightly affected whereas the cost of the random experiments will increase. Therefore, we expect the effectiveness of the algorithm to increase as circuit size increases.

2.9 Miscellaneous issues

The main disadvantage of our embedding scheme, as presented in the paper, is the fact that it is applicable only to a limited family of circuits, namely those for which the hardest fault has less than 2^δ test patterns and for which $L < 2^{max}$.

While these two parameters are self imposed by each individual user, they still limit the family of circuits. By allowing for higher values of δ and max, the family increases, at a cost of more preprocessing effort and longer test sessions.

The major limiting factor, besides 100% fault coverage, is the requirement of one seed and no reconfiguration of the hardware. These requirements were set so as to limit to a bare minimum the hardware and delay overhead of the BIST circuitry. By relaxing these restrictions, some tradeoffs arise.

2.9.1 Multiple seeds

First we consider the use of multiple seeds for the LFSR. A number of complications arise in this scenario.

First, this will require higher control complexity of the BIST circuitry to allow multiple seeding at the desired points in time.

A more subtle complication is that by using multiple seeds, the overall test length, i.e. total number of patterns that are applied to the CUT, is reduced, in some cases drastically. This affects the probability of detecting the *easy* faults. The higher the reduction in test length, the higher the decrease in detection probability, resulting

in a need to embed test patterns for more faults. This, in turn, would increase the preprocessing effort.

A third problem is the computation complexity of finding the optimal subsequence for an s-seed test sequence. This problem is $NP-hard$, hence exponential in the number of seeds.

To show that the problem is NPH we consider the NPC problem *minimum cover* [15, p. 222]. The instance of the problem is a collection C of subsets of a finite set S, and a positive integer $K < |C|$. The question asked is whether exists a subset $C' \subset C$ with $|C'| \leq K$ such that every element of S belongs to at least one element of C'?

The reduction to our problem is as follows. Let S be the set of faults. Let $C = \{c_1, \ldots, c_{|C|}\}$ where each c_i is the set of faults detected by the i-th test pattern, when sorted according to their logarithms. We assume the distance between consecutive test patterns is K. We apply our algorithm that finds the optimal selection of K seeds to detect all faults. If the resulting test sequence length is at most K, then each seed was used as a test pattern and immediately changed, i.e. the pattern generator was not allowed to run in autonomous mode. This is due to the fact that in autonomous mode, from test pattern i to test pattern $i+1$ $K+1$ test pattern are applied to the circuit. Thus, if the optimal test sequence is of length at most K, there exists a cover for S with $|C'| \leq K$ and C' is the subsets that are represented by the selected seeds.

2.9.2 Lower fault coverage

A second thought is to lower the fault coverage requirement. This can easily be achieved by modifying the windowing procedure to consider subsequence that detects the desired percentage of the *hard* faults. The result will be shorter test sequences, which will lower the probability of detecting some of the *easy* faults.

There are two problems with this scheme. The first is what percentage of the *hard* faults need to be detected to guarantee a required overall coverage, i.e. if we

require an overall fault coverage of $y\%$, what is the percentage of the *hard* faults that need to be embedded?

The second issue is the effectiveness of the embedding procedure for certain values of y, i.e. is the reduced effort in computation worth the resulting reduction in test length? As y decreases the required test length also decreases and a random selection is more likely to achieve the required coverage with a competitive length.

2.9.3 Embedding subsets of test patterns

When the complete sets for the *hard* faults are too large, one might consider embedding only a subset of the test patterns for these faults. The analysis in Section 2.1, regarding the probability of not detecting a given fault, is really the probability of not generating any pattern out of a given set of patterns. What matters in the analysis is the size of the smallest set of patterns, and the parameter L is relative to this size. The length of the embedded sequences is relative to L, and by empirical observations it is between $L/2$ and L. Therefore, we expect that when given only subsets of the test patterns, the embedded sequence length will be longer than necessary, the ratio inversely proportional to the size of the subset relative to the full set. If this ratio is greater than 11, then we achieved no improvement over the recommended test length of [39].

2.9.4 Using wider pattern generators

Wagner et al. [45] looked at the question of the required test length when the PG width is wider than the input width. This has the effect of increasing the number of possible inputs and the number of test patterns (N and K respectively) by the same factor, hence the expected test length (L) is unchanged. From our perspective the resulting embedded sequence length remains the same but the preprocessing work (number of logarithm computations per hard fault) increases. Thus, there is no advantage to using wider *PG*s.

2.9.5 Different windows from one polynomial

By considering different windows from the same polynomial, the preprocessing work can be reduced without paying too much in sequence length. If the smallest window for a given polynomial turns out not to achieve 100% fault coverage, try the second best window and so forth until a satisfying window is found. By our experimental results, for six of thirteen circuits the worse embedded sequence was still better than the best RS sequence. For one circuit there was only one embedded sequence and for the remaining seven circuits, the ratio of embedded length versus RS lenght was 1.72, 1.51, and the rest were less than 1.21. For circuits *exep* and *in*5 the RS experiments took less time than embedding with one polynomial, but for the remaining circuits the ratio of RS time versus embedding time for one polynomial ranged from 1.7 to over 60.

Chapter 3

Response analyzer design

The compacting function of a MISR is polynomial division over $GF[2]$. The *effective output polynomial* is divided by the feedback polynomial. The signature is the remainder of the division.

Our objective is to select a feedback polynomial for the compacting MISR, given a set of modeled faults, such that an erroneous response resulting from any modeled fault is mapped to a different signature than the good response.

We assume the following test scenario. The input sequence to the CUT has been designed so that the effective output polynomial due to any target fault is different than the effective polynomial of the good response. Let r be the effective polynomial of the good response, then the effective polynomial due to fault i can be represent as $r + h_i$. By the linearity of the remaindering operation, we get a different remainder for this erroneous polynomial iff h_i is not divisible by the feedback polynomial. We assume we are given the error polynomials for each of the target faults.

The problem we deal with in this chapter is the following: given a set of polynomials $H = \{h_1, h_2, \ldots, h_{|H|}\}$ find a polynomial that is relatively prime to all the polynomials of H. Such a polynomial will be referred to as a *non-factor* of H. If a non-factor is used as the feedback polynomial for the compacting MISR, zero-aliasing is achieved for the set of target faults. In particular, for irreducible and primitive feedback polynomials we present (1) upper bounds on the smallest degree zero-aliasing MISR; (2) procedures for selecting a zero-aliasing MISR with the

smallest degree; (3) procedures for selecting a zero-aliasing MISR of a pre-specified degree (under the condition that one exists); and (4) procedures for fast selection of a zero-aliasing MISR. We analyze the worst case as well as expected time complexity of the proposed procedures.

A note on notation. The polynomials $\{h_i\}$ represent the error polynomials. The degree of h_i is represented by d_i. The product of the polynomials in H is denoted by h, and the degree of h is d_h. For each h_i, the product of the distinct irreducible factors of degree j of h_i is denoted by $g_{i,j}$, with $d_{i,j}$ being the degree of $g_{i,j}$. The product, over all i, of the polynomials $g_{i,j}$ is denoted by g_j. The non-factor we seek will be referred to as a with d_a representing the degree of a.

The rest of this chapter is organized as follows. In Section 3.1 we establish upper bounds on the degree of a non-factor. In Section 3.2 we review polynomial operations over $GF[2]$ and their complexities. Section 3.3 presents procedures for finding a non-factor of smallest degree for the set H. Section 3.4 presents procedures for finding a non-factor of a pre-specified degree and for finding a non-factor fast. We also discuss the effectiveness of conducting an exhaustive search for a least degree non-factor. Section 3.5 presents some experimental data. We conclude with Section 3.6.

3.1 Bounds on the least degree non-factor of a set of polynomials

Consider the following problem.

Problem 1: Let H be a set of $|H|$ polynomials $h_1, \ldots, h_{|H|}$ with $deg(h_i) = d_i$ and for all $1 \leq i \leq |H|$, $d_i \leq n$. Let $h = \prod_{i=1}^{|H|} h_i$. Then $deg(h) = \sum_{i=1}^{|H|} d_i = d_h \leq |H|n$. Give an upper bound $s(d_h)$ on the degree of an irreducible polynomial and an upper bound $p(d_h)$ on the degree of a primitive polynomial that does not divide h, i.e. there exists an irreducible (primitive) polynomial of degree at most $s(d_h)$ ($p(d_h)$) that does not divide h. □

Similarly, let $es(H)$ ($ep(H)$) be the *expected* degree of an irreducible (primitive) polynomial that is a non-factor of H.

The bounds $s(d)$ and $p(d)$ will be referred to as the *worst case bounds* while the bounds $es(H)$ and $ep(H)$ will be referred to as the *expected bounds.* We first establish the worst case bounds and then proceed with the expected bounds.

3.1.1 The worst case bounds

For the bound on $s(d)$ we follow [24]. Let $I_2(j)$ denote the number of irreducible polynomials of degree j over $GF[2]$. The degree of the product of all irreducible polynomials of degree j is $jI_2(j)$. Let $s(d)$ denote the least integer such that $\sum_{j=1}^{s(d)} jI_2(j) > d$. Let $Q_{s(d)}$ be the product of all the irreducible polynomials of degree less than or equal to $s(d)$. The degree of $Q_{s(d)}$ is greater than d. Replacing d with d_h, $Q_{s(d_h)}$ has at least one root that is not a root of h, hence $Q_{s(d_h)}$ has at least one irreducible factor that is not a factor of h. Thus, $s(d_h)$ is an upper bound on the degree of an irreducible polynomial that is relatively prime to all the polynomials in the set H. The following lemma provides a bound on $s(d_h)$.

Lemma 3: [24, Lemma 4, p.293]

$$s(d) \leq \lceil \log(d+1) \rceil$$

□

We turn to find the bound on $p(d)$. The number of primitive polynomials of degree m over $GF[2]$ is

$$\frac{1}{m}\phi(2^m - 1)$$

where $\phi(q)$ is the *Euler function* denoting the number of integers less than and relatively prime to q and ([25, p. 37])

$$\phi(q) = q \prod_{i=1}^{l} (1 - 1/p_i)$$

where the p_i's are all the distinct prime factors of q.

Lemma 4: [35, p. 173]

$$\phi(q) \geq e^{-\gamma} \frac{q}{\ln \ln q + \frac{5}{2e^{\gamma} \ln \ln q}}$$

for all $q \geq 3$ with the only exception being $q = 223,092,870$ (the product of the first nine primes), for which $\frac{5}{2}$ is replaced by 2.50637. $\gamma = 0.577215665\ldots$ is *Euler's constant.* □

For $q > 65$ we have

$$\phi(q) > \frac{q}{3 \ln \ln q} > \frac{q}{2.08 \log \log q}. \tag{3.1}$$

To help us derive the bound on $p(d)$ we introduce the value $\tau(t)$. Let $\tau(t)$ denote the least integer such that the ratio between $\tau(t)$ times the number of primitive polynomials of degree $\tau(t)$ and t times the number of irreducible polynomials of degree t is greater than 1, i.e.

$$\frac{\phi(2^{\tau(t)} - 1)}{t I_2(t)} > 1.$$

Lemma 5: [24, Lemma 3, p. 293] For $t \geq 3$

$$2^{t-1} < t I_2(t) \leq 2^t - 2.$$

□

Lemma 6: For $t > 6$, $\tau(t) \leq t + \lceil 1 + \log \log 2t \rceil$

Proof: For $q \geq 7$, the function $\frac{q}{2.08 \log \log q}$ is an increasing function. Also,

$$\frac{q}{2.08 \log \log q} > \frac{q-1}{2.08 \log \log q} \quad \text{and} \quad \frac{q-1}{2.08 \log \log (q-1)} > \frac{q-1}{2.08 \log \log q}$$

hence,

$$\frac{q-1}{2.08\log\log(q-1)}+1>\frac{q}{2.08\log\log q}.$$

Let $\tau'(t)$ be the least integer such that $\frac{2^{\tau'(t)}}{2.08\log\log 2^{\tau'(t)}} > 2^t$. Thus, using the above relation,

$$\frac{2^{\tau'(t)}-1}{2.08\log\log(2^{\tau'(t)}-1)} > 2^t-1.$$

By Equation (3.1) we get

$$\phi(2^{\tau'(t)}-1) > 2^t-1$$

and due to Lemma 5

$$\phi(2^{\tau'(t)}-1) > tI_2(t).$$

Thus, by the definition of $\tau(t)$, we have that $\tau'(t) \geq \tau(t)$. To bound $\tau(t)$ from above, we solve for $\tau'(t)$.

By definition, $\tau'(t)$ must satisfy

$$\begin{array}{lcll} \frac{2^{\tau'(t)}}{2.08\log\tau'(t)} & > & 2^t & \Rightarrow \\ 2^{\tau'(t)-t} & > & 2.08\log\tau'(t) & \Rightarrow \\ \tau'(t)-t & > & \log 2.08+\log\log\tau'(t) & \end{array}$$

By setting $\tau'(t) = t + \lceil 1+\log\log 2t\rceil$, we have

$$\tau'(t)-t = \lceil 1+\log\log(2t)\rceil$$

and for $t > 6$

$$\lceil 1+\log\log(2t)\rceil > \log 2.08+\log\log\tau'(t)$$

$\square$

Lemma 7: Let $p(d)$ denote the least integer such that $\sum_{j=1}^{p(d)}\phi(2^j-1) > d$, then for $d > 65$

$$p(d) \leq \lceil\log(d+1)\rceil + \lceil 1+\log\log(2\lceil\log(d+1)\rceil)\rceil.$$

Table 3.1: Number of primitive elements in $GF[2^m]$ and accumulated number of primitive elements in $GF[2^2]\ldots GF[2^m]$

m	$\phi(2^m-1)$	$\sum_{i=2}^{m}\phi(2^i-1)$	m	$\phi(2^m-1)$	$\sum_{i=2}^{m}\phi(2^i-1)$
2	2	2	28	132,765,696	343,973,802
3	6	8	29	533,826,432	877,800,234
4	8	16	30	534,600,000	1,412,400,234
5	30	46	31	2,147,483,646	3,559,883,880
6	36	82	32	2,147,483,648	5,707,367,528
7	126	208	33	6,963,536,448	12,670,903,976
8	128	336	34	11,452,896,600	24,123,800,576
9	432	768	35	32,524,320,000	56,648,432,576
10	600	1,368	36	26,121,388,032	82,769,820,608
11	1,936	3,304	37	136,822,635,072	219,592,455,680
12	1,728	5,032	38	183,250,539,864	402,842,995,544
13	8,190	13,222	39	465,193,834,560	868,036,830,104
14	10,584	23,806	40	473,702,400,000	1,341,739,230,104
15	27,000	50,806	41	2,198,858,730,832	3,540,597,960,936
16	32,768	83,574	42	2,427,720,325,632	5,968,318,286,568
17	131,070	214,644	43	8,774,777,333,880	14,743,095,620,448
18	139,968	354,612	44	8,834,232,287,232	23,577,327,907,680
19	524,286	878,898	45	28,548,223,200,000	52,125,551,107,680
20	480,000	1,358,898	46	45,914,084,232,320	98,039,635,340,000
21	1,778,112	3,137,010	47	140,646,443,289,600	238,686,078,629,600
22	2,640,704	5,777,714	48	109,586,090,557,440	348,272,169,187,040
23	8,210,080	13,987,794	49	558,517,276,622,592	906,789,445,809,632
24	6,635,520	20,623,314	50	656,100,000,000,000	1,562,889,445,809,632
25	32,400,000	53,023,314	51	1,910,296,842,179,040	3,473,186,287,988,672
26	44,717,400	97,740,714	52	2,338,996,194,662,400	5,812,182,482,651,072
27	113,467,392	211,208,106	53	9,005,653,101,120,000	14,817,835,583,771,072

Proof: By the definition of $\tau(t)$ and Lemma 6

$$\sum_{j=1}^{j=\lceil\log(d+1)\rceil+\lceil 1+\log\log(2\lceil\log(d+1)\rceil)\rceil}\phi(2^j-1) > \sum_{j=1}^{j=\lceil\log(d+1)\rceil} jI_2(j).$$

By Lemma 3 and the definition of $s(d)$

$$\sum_{j=1}^{j=\lceil\log(d+1)\rceil} jI_2(j) \geq \sum_{j=1}^{s(d)} jI_2(j) > d.$$

$\square$

Example 1: In Table 3.1 the values of $\phi(2^m - 1)$ (the degree of the product of all the primitive polynomials of degree m) and $\sum_{i=2}^{m} \phi(2^i - 1)$ (the degree of the product of all the primitive polynomials of degree $2 \leq i \leq m$) are tabulated for $2 \leq m \leq 53$. As long as d is less than the maximum value in the table, $p(d)$ can be read from the table, instead of using Lemma 7. For example, if the number of modeled faults in the CUT is $|H| = 10^4$ and the length of the test sequence is $n = 10^6$, then $d_h \leq 10^{10}$. The degree of the product of all primitive polynomials with degree less than or equal to 33 is the first which is greater than 10^{10}, hence $p(d_h) \leq 33$. Thus, a zero-aliasing LFSR with a primitive feedback polynomial, of degree at most 33, exists for the CUT. On the other hand, using the bound of Lemma 7 we get $p(d_h) \leq 38$. □

A closer look at Table 3.1 shows that the product of all the primitive polynomials of degree less than or equal to 53 has degree D greater than $1.4 \cdot 10^{16}$. Thus, as long as the product of the number of faults and the test sequence length is less than D (which is the case for all practical test applications) a zero-aliasing MISR of degree less than or equal to 53 exists.

3.1.2 The expected bounds

In deriving the expected bounds we assume that the polynomials $\{h_i\}$ are random polynomials. Denote the product of the distinct irreducible factors of degree j of h_i by $g_{i,j}$. Denote the number of distinct irreducible factors of h_i, of degree j, by v. The value of v can range from 0 to $min\{\lfloor d_i/j \rfloor, I_2(j)\}$.

Lemma 8: For $j \geq 2$, the *expected value* of v (the number of irreducible factors of $g_{i,j}$) is less than or equal to $\frac{1}{j}$.

Proof: Let $IR_2(j) = \{p_i\}_{i=1}^{I_2(j)}$ be the set of irreducible polynomials of degree j over $GF[2]$. For a given polynomial q, of degree greater or equal to j, define the *indicator function* $d(p_i, q)$ to be one if p_i divides q and zero otherwise. The probability that a

polynomial of degree j divides a random polynomial of degree greater or equal to j is 2^{-j}, hence the probability that $d(p_i, q) = 1$ is equal to 2^{-j}. Thus

$$\begin{aligned} E[v] &= E\left[\sum_{i=1}^{I_2(j)} d(p_i, q)\right] \\ &= \frac{I_2(j)}{2^j} \\ &< \frac{1}{j} \end{aligned}$$

□

The same type of analysis can be used to bound $Var[v]$, the variance of v, and σ_v, the standard deviation of v.

Lemma 9: For $j \geq 2$, the *variance* of the number of irreducible factors of $g_{i,j}$ is less than $\frac{2}{j}$. The *standard deviation* is less than $\left(\frac{2}{j}\right)^{\frac{1}{2}}$.

Proof: The variance of v is given by $Var[v] = E[v^2] - E[v]^2$.

$$\begin{aligned} E[v^2] &= E\left[\left(\sum_{i=1}^{I_2(j)} d(p_i, q)\right)^2\right] \\ &= E\left[\sum_{i=1}^{I_2(j)} d(p_i, q)^2 + 2 \sum_{\substack{p_i, p_k \in IR_2(j) \\ i<k}} d(p_i, q) d(p_k, q)\right] \\ &= \sum_{i=1}^{I_2(j)} E\left[d(p_i, q)^2\right] + 2 \sum_{\substack{p_i, p_k \in IR_2(j) \\ i<k}} E\left[d(p_i, q) d(p_k, q)\right] \\ &< \frac{I_2(j)}{2^j} + \frac{2 I_2(j)^2}{2^{2j}} \\ &< \frac{1}{j} + \frac{2}{j^2}. \end{aligned}$$

For $j \geq 2$ we have

$$E[v^2] < \frac{2}{j}$$

and,

$$Var[v] = E[v^2] - E[v]^2 < E[v^2] < \frac{2}{j} \text{ and } \sigma_v < \left(\frac{2}{j}\right)^{\frac{1}{2}}.$$

□

Having computed the mean and variance of the number of irreducible factors of degree j per polynomial, we can compute a *confidence measure* for these results.

Lemma 10: For $j \geq 8$, the expected number of polynomials $g_{i,j}$ with more than 5 (50) factors is less than $|H|/100$ ($|H|/10,000$).

Proof: Using the *Chebyshev inequality* [16, p. 376]

$$Pr(|v - E[v]| \geq c\sigma_v) \leq 1/c^2$$

for $j \geq 8$ the probability that v is greater than 5 is less than 0.01. Using this result we can define a second random process in which the random variable x is 1 iff v is greater than 5 and 0 otherwise. This process is a *Bernoulli experiment* [10, Sec. 6.4]. The expected number of $g_{i,j}$'s with more than 5 factors is upper bounded by $|H|/100$, as is the variance. Similarly, the probability that v is greater than 50 is less than 0.0001 and the expected number of $g_{i,j}$'s with more than 50 factors is bounded by $|H|/10,000$. □

Lemma 11: The expected degree of the smallest irreducible non-factor of the set of polynomials H is bounded from above by $\lceil \log |H| \rceil + 1$.

Proof: Denote the product of the polynomials $g_{i,j}$, $1 \leq i \leq |H|$, by g_j. By Lemma 8, the expected number of (not necessarily distinct) factors of g_j is less than $|H|/j$. The smallest j for which $I_2(j)$ exceeds this value is an upper bound on the expected degree d_a of a non-factor of H. Thus, the upper bound δ on d_a satisfies

$$\frac{|H|}{\delta} < \frac{2^{\delta-1}}{\delta} < I_2(\delta) \;\Rightarrow\; |H| < 2^{\delta-1}$$

and $d_a \leq \lceil \log |H| \rceil + 1$. □

By applying Lemma 6 on the result of Lemma 11, we have

Corollary 12: The expected degree of the smallest primitive non-factor of the set of polynomials H is bounded from above by $2 + \lceil \log |H| \rceil + \lceil \log \log(2 + 2\lceil \log |H| \rceil) \rceil$. □

Example 2: Using the numbers of Example 1, let $|H| = 10^4$ and $n = 10^6$. The first j for which $\phi(2^j - 1)/j$ exceeds $|H|/j$ is $j = 14$ (Table 3.1), hence we expect to find a zero-aliasing MISR with a primitive feedback polynomial of degree less than or equal to 14, as opposed to the worst case of 33. Corollary 12 would give us an upper bound of 19. □

As the expected bound is a only a function the number of faults and not the length of the test sequence, the expected degree of a zero-aliasing MISR will *never* exceed 53. In fact, as long as the number of faults is less than 1 million, we expect to find a zero-aliasing MISR of degree less than or equal to 21.

3.2 Polynomial operations in $GF[2]$

In search for a (least degree) non-factor of H we use procedures that sift the factors of the same degree from a given polynomial. These procedures are based on the following lemma.

Lemma 13: [25, Lemma 2.13, p.48] $x^{2^m} - x$ is the product of all irreducible polynomials of degree l, where l is a divisor of m. □

Thus, a basic step in finding the distinct irreducible factors of a polynomial $b(x)$ is the computation of

$$g(x) = gcd(b(x), x^{2^m} - x).$$

The result of this operation is the product of all the irreducible factors of degree l, where $l|m$, of $b(x)$. For most polynomials $b(x)$ of interest to us, $2^m >> deg(b(x))$. Therefore, we first compute

$$r(x) \equiv (x^{2^m} - x) \bmod b(x)$$

and then

$$g(x) = gcd(b(x), r(x)).$$

We first discuss the complexity of polynomial operations in $GF[2]$ and then review a well known approach to reduce x^{2^m} modulo $b(x)$.

3.2.1 Polynomial multiplication, division and gcd

The complexity of a polynomial *gcd* operation is $O(M(s) \log s)$ [3, pp. 300-308], where s is the degree of the larger polynomial operand and $M(s)$ is the complexity of polynomial multiplication, where the product has degree s. The complexity of polynomial division is also $O(M(s))$ [3, Ch. 8]. Hence, it is crucial to find an efficient multiplication algorithm.

We consider two multiplication algorithms. Both algorithms are based on FFT techniques [3, Ch. 7], [10, Ch. 32]. For these algorithms to work they need a root of unity whose order has small prime factors. In most cases, when the product polynomial has degree s, a root of order $2^m > s$ is used. This poses a problem, since fields of characteristic 2 do not contain such roots.

The first algorithm is due to Schönhage [41]. It uses roots of order 3^{m+1} to multiply polynomials of degree $s < 3^m$. Its complexity is $O(s \log s \log\log s)$.

The second algorithm is suggested by Cormen et al. [10, p. 799]. To multiply polynomials of degree $s/2 < 2^{m-1}$ they suggest working in the field $GF[p]$ where p is a prime of the form $\beta \cdot 2^m + 1$. The multiplication is done over $GF[p]$ and the coefficients of the product are reduced modulo 2 to give the correct result over $GF[2]$. The question that naturally arises is how big is p. The best provable bound on the size of p is that it is less than $2^{5.5 \cdot m}$ [19]. It is widely believed that $\beta = O(m^2)$ [35, p. 221]. A detailed discussion on the bounds on p, as well as how to find the least prime p and the smallest primitive element in $GF[p]$ is given in Appendix A.

The complexity, per multiplication, of the Cormen algorithm is $O(s \log s)$ operations in $GF[p]$. If the word size of a machine is greater than $\log p$, then word operations can be performed in $O(1)$ machine instructions. To further cut down on

Table 3.2: Least prime p, of the form $\beta 2^m + 1$, with smallest generator α and 2^m-th root of unity ω.

m	p	α	ω	m	p	α	ω	m	p	α	ω
6	193	5	11	11	12289	11	7	16	65537	3	3
7	257	3	9	12	12289	11	41	17	786433	10	8
8	257	3	3	13	40961	3	12	18	786433	10	5
9	7681	17	62	14	65537	3	15	19	5767169	3	12
10	12289	11	49	15	65537	3	9	20	7340033	3	5

time, we construct logarithmic tables relative to a primitive element, α, of $GF[p]$ for multiplication and addition. With these tables, multiplication modulo p is addition modulo p of the logarithms. The addition table stores the *Jacobi logarithm* $Z(i)$ [25, p. 69], i.e. $Z(i) = \log_\alpha(\alpha^i + 1)$ where $\log_\alpha(0)$ is defined as ∞. Thus, adding $\alpha^\mu + \alpha^\nu$, for $\mu \leq \nu$, is actually the multiplication operation $\alpha^\mu(1 + \alpha^{\nu-\mu})$ and the logarithm of the result equals $\mu + Z(\nu - \mu)$.

Table 3.2 shows the smallest p for $m = 6, 7, \ldots, 20$, along with the smallest primitive element α and the smallest $2^m - th$ root of unity ω.

In the sequel we shall use the notation $O(M(s))$ for the complexity of polynomial multiplication. Whenever possible it will mean $s \log s$, otherwise it should be taken as $s \log s \log\log s$. Similary the notation $L(s)$ will denote either $\log s$ or $\log s \log\log s$, as appropriate.

3.2.2 x^{2^m} modulo $b(x)$ and x^t modulo $b(x)$

We review a well known approach [5] [33] to find the remainder of x^{2^m} when divided by $b(x)$ without actually carrying out the division.

Let $s = deg(b(x))$ and let $R^{(j)}(x) = \sum_{i=0}^{s-1} R_i^{(j)} x^i \equiv x^{2^j} \bmod b(x)$. Then

$$R^{(j)^2}(x) \equiv (x^{2^j})^2 \bmod b(x) \equiv x^{2^{j+1}} \bmod b(x) = R^{(j+1)}(x).$$

Squaring over $GF[2]$ is easy ($r(x)^2 = r(x^2)$), thus by repeatedly squaring and reducing modulo $b(x)$, in time $O(mM(s))$ we can compute $R^{(m)}(x)$. Note that the maximum degree $R^{(j)^2}(x)$ can have is $2(s-1)$. Once we have $R^{(m)}(x)$, we can compute $g(x) = gcd(b(x), R^{(m)}(x) - x)$ in time $O(M(s)\log s)$. Overall time needed to compute $g(x)$ is $O(mM(s) + M(s)\log s)$.

Let $t = \sum_{j=0}^{m} t_j 2^j$. To compute $r(x) = x^t \bmod b(x)$ we compute $R^{(m)}(x)$ as described above. This costs $O(mM(s))$. We initialize $r(x)$ to equal 1. As we compute $R^{(m)}(x)$, for each intermediate value $R^{(j)}(x)$ for which $t_j = 1$, we set $r(x)$ to the product $r(x)R^{(j)}(x) \bmod b(x)$. Each such computation costs $O(M(s))$, hence the cost of computing $x^t \bmod b(x)$ is $O(mM(s))$.

3.3 Finding a non-factor of smallest degree for a given set of polynomials

After establishing the bounds on the least degree non-factor of H in Section 3.1, this section addresses the question of finding a least degree non-factor for H.

Problem 2: Given a set of polynomials $H = \left\{h_1(x), h_2(x), \ldots, h_{|H|}(x)\right\}$ with $deg(h_i) = d_i \leq n$, let

$$h(x) = \prod_{i=1}^{|H|} h_i(x) \ , \ \ deg(h) = \sum_{i=1}^{|H|} d_i = d_h.$$

Find an irreducible (primitive) polynomial $a(x)$, with $deg(a) = d_a$, such that

1. For all $1 \leq i \leq |H|$, $h_i \not\equiv 0 \bmod a$ (equivalently, $h \not\equiv 0 \bmod a$).
2. For all irreducible (primitive) polynomials $b(x)$, with $deg(b) < d_a$, $h \equiv 0 \bmod b$ (or equivalently, there exists an i for which $h_i \equiv 0 \bmod b$).

□

One way of solving the problem is by factoring the polynomials of H. This would require too much work, since we do not need to know all the factors in order to find a non-factor. We only need to know the "small" factors.

In this section we present algorithms for solving Problem 2 and analyze their complexity. The complexity is given in two forms. The first is with *worst case complexity bounds*, referred to as the *worst case complexity*. The second is with *expected complexity bounds*, referred to as the *expected complexity*. The expected complexity is a refinement of the worst case complexity based on the *expected size* of the results from our procedures.

By Lemmas 3 and 7 (Section 3.1), we have an upper bound $u = s(d_h)$ or $u = p(d_h)$ on d_a, depending on whether we are looking for an irreducible or a primitive non-factor. Using this bound, we begin our search process, which is made up of three phases.

1. For all $h_i \in H$, find $g_{i,j}(x)$, the product of all distinct irreducible (primitive) factors of h_i, of degree j.

2. Having found the polynomials $g_{i,j}$, determine whether all irreducible (primitive) polynomials of degree j are factors of H.

3. If not all irreducible (primitive) polynomials of degree j are factors of H, find one that is not.

The worst case complexities of the three phases for the irreducible case are $O(|H|u^2M(n))$, $O(|H|^2M(n)\log n)$ and $O(|H|^2n^2u^2M(u))$. The dominant term is $O(|H|^2n^2u^2M(u))$. The worst case complexities of the three phases for the primitive case are $O(|H|u^3M(n))$, $O(|H|^2 \cdot M(n)\log n)$ and $O(|H|^2n^2u^3M(u)\log\log u)$. The dominant term is $O(|H|^2n^2u^3M(u)\log\log u)$.

The expected complexity of the first two phases are $O(|H|u^2M(n))$ and $O(|H| \cdot \log|H| \cdot u^2L(n)\log n)$. The expected complexity for the third phase is $O(|H|\log|H| \cdot d_aM(d_a))$ to find an irreducible non-factor and $O(|H|\log|H|d_a^2\log\log d_aM(d_a))$ to find a primitive non-factor. The dominant term is $O(|H|u^2M(n))$.

The worst case complexity is a function of $|H|^2 n^2$ multiplied by terms that are logarithmic in $|H|$ and n whereas the expected complexity is a function of $|H|n$ multiplied by terms that are logarithmic in $|H|$ and n.

3.3.1 The product of all distinct factors of the same degree for a given polynomial

Given the polynomial $h_i(x)$ and the upper bound u, we wish to compute $g_{i,j}$, the product of all distinct factors of h_i of degree j, for $1 \leq j \leq u$. The procedure for computing the polynomials $g_{i,j}$ is given in Figure 3.1. The polynomials $g_{i,j}$ are computed in three steps. First, for $u/2 < j \leq u$, compute $g_{i,j} = gcd(h_i(x), x^{2^j} - x)$. Each $g_{i,j}$ is a product of all the distinct irreducible factors of $h_i(x)$ of degree j and of degree l, where $l|j$.

When j is less than or equal to $u/2$, we have $2j \leq u$. By Theorem 13, $g_{i,2j}$ contains the product of all irreducible factors of degree l, where $l|j$, of h_i. Since the degree of $g_{i,2j}$ is (much) less than the degree of h_i, it is more efficient to compute $g_{i,j}$ from $g_{i,2j}$ than from h_i. Thus, in Step 2, for $1 < j \leq u/2$ compute $g_{i,j} = gcd(g_{i,2j}, x^{2^j} - x)$.

At the end of Step 2, each $g_{i,j}$ contains *all* the factors of degree $l|j$ of h_i. To sift out the factors of degree less than j from $g_{i,j}$, we need to divide $g_{i,j}$ by $g_{i,l}$, where l ranges over the set of divisors of j. This is carried out in Step 3.

Procedure *distinct_factors*() is not enough when we are looking for a primitive non-factor. At the end of the procedure, each $g_{i,j}$ is the product of all distinct irreducible polynomials of degree j, that are factors of h_i. From $g_{i,j}$ we need to sift out the non-primitive factors. Before describing this aspect, we introduce the notion of *maximal divisors*.

Definition 1: Let $q = \prod_{i=1}^{r} p_i^{e_i}$, with $p_1, \ldots, p_r$ being the distinct prime factors of q. The set of *maximal divisor* of q is the set $md(q) = \{m_i\}_{i=1}^{r}$ where $m_i = q/p_i$. $\square$

For example, $20 = 2^2 \cdot 5$, hence $md(20) = \{10, 4\}$. $16 = 2^4$, therefore $md(16) = \{8\}$. Since 7 has only one prime factor, $md(7) = \{1\}$.

Procedure 3: *distinct_factors(h_i)*

1. For $(j = u\,; j > u/2\,; j--)$
   (a) $g_{i,j} = gcd(h_i(x), x^{2^j} - x)$
2. For $(j = \lfloor u/2 \rfloor\,; j > 0\,; j--)$
   (a) $g_{i,j} = gcd(g_{i,2j}, x^{2^j} - x)$
3. For $(j = 2; j \leq u; j++)$
   (a) For all $l|j$
      i. $g_{i,j} = g_{i,j}/g_{i,l}$

Figure 3.1: Procedure $distinct_factors(h_i)$. Computes the product of all distinct factors of degree j, for $1 \leq j \leq u$, of the polynomial h.

Procedure 4: *distinct_primitive($g_{i,j}$)*
/* sifts out the non-primitive factors of $g_{i,j}$ */

1. For all l in $md(2^j - 1)$
   (a) $q_l = gcd(g_{i,j}, x^l - 1)$
   (b) $g_{i,j} = g_{i,j}/q_l$

Figure 3.2: Procedure $distinct_primitive(g_{i,j})$. Sifts out the non-primitive factors of $g_{i,j}$.

A polynomial over $GF[q]$ of degree m is irreducible iff it divides $x^{q^m-1} - 1$ and does not divide $x^{q^k-1} - 1$ for all divisors k of m. It is primitive of degree m iff it is irreducible and does not divide $x^l - 1$ for all l in $md(q^m - 1)$ [25, Ch. 3]. Procedure *distinct_primitives*(), shown in Figure 3.2, sifts out the non-primitive factors of $g_{i,j}$.

Lemma 14:

1. The complexity of Procedure *distinct_factors*() is $O(u^2 M(n))$.
2. The complexity of Procedure *distinct_primitive*() is $O(u^3 M(n))$.

3. The complexity of the first phase is $O(|H|u^2M(n))$ for the irreducible case and $O(|H|u^3 \cdot M(n))$ for the primitive case.

In the above expressions $u = s(d_h)$ for the irreducible case and $u = p(d_h)$ for the primitive case.

Proof:

1. The worst case complexity of Procedure *distinct_factors*() is as follows. In Step 1, the procedure performs $u/2$ *gcd* computations involving h_i. The complexity of each *gcd* computation is $O(jM(d_i) + M(d_i)\log d_i)$. Thus the total work for the first stage is

$$\sum_{j=\lceil u/2\rceil}^{u} O(jM(d_i) + M(d_i)\log d_i) = O\left(\frac{u}{2}M(d_i)\left(\frac{3u}{4} + \log d_i\right)\right) = O(u^2M(n)).$$

   In Step 2 the procedure carries out $u/2$ *gcd* operations. The work required for this step is

$$\sum_{j=1}^{\lfloor u/2\rfloor} O(jM(d_{i,2j}) + M(d_{i,2j})\log d_{i,2j}) = O(u^2M(n)).$$

   In Step 3, for every element of the sets of divisors, the procedure performs a division operation. The cost expression is

$$\sum_{j=1}^{u}\sum_{l|j} O(M(d_{i,j})) < \sum_{j=1}^{u} O(jM(d_{i,j})) = O(u^2M(n)).$$

2. The complexity of Procedure *distinct_primitive*() is as follows. Each iteration of Procedure *distinct_primitives*() reduces $(x^k - 1)$ modulo $g_{i,j}$ and performs one *gcd* and one division operation. The cost of each iteration is $O(jM(d_{i,j}) + M(d_{i,j})\log(d_{i,j}))$. There are $md(2^j - 1)$ iterations, with $md(2^j - 1) < j \leq u$, and we run the procedure u times. Therefore, the additional work for the primitive case is bounded by $O(u^3M(d_i))$.

In most cases, the values $d_{i,j}$ will be (much) less than n, hence the actual work will be much less then $O(u^2 M(n))$ and the dominant factor will be Step 1 of Procedure *distinct_factors*().

3. Over the set H, based on 1 and 2, the complexity of the first phase is $(|H|u^2 \cdot M(n))$ for the irreducible case and $(|H|u^3 M(n))$ for the primitive case. The value of u is either $s(d_h)$ or $p(d_h)$ corresponding to either the irreducible or primitive case.

□

Lemma 15: The expected complexity of the first phase is $O(|H|u^2 M(n)))$ with u equal to either $es(H)$ or $ep(H)$.

Proof: The expected complexity of Procedure *distinct_factors*() is dominated by the complexity of Step 1, which is $O(u^2 M(n))$. The difference in the complexity of the other steps, over the worst case, comes from using the expected size of the $d_{i,j}$s, instead of their worst case size, which is equal to n. The expected complexity of the procedure (including Procedure *distinct_primitive*()), over the set H is, thus, $O(|H|u^2 M(n))$, with u equal to either $es(H)$ or $ep(H)$. □

3.3.2 The number of all distinct factors, of the same degree, for a set of polynomials

After the first phase, for all degrees $1 \leq j \leq u$, we have $|H|$ polynomials $g_{i,j}$, each a product of the distinct irreducible (primitive) factors of degree j of h_i. Some of the $g_{i,j}$'s might equal 1 while some pairs might have factors in common. Our goal is to find a least degree non-factor of H. The first thing we must determine is whether all irreducible polynomials of degree j appear in $g_j = \prod_i^{|H|} g_{i,j}$. This is the second of our three phases (page 72). A simple test is to compare $\frac{deg(g_j)}{j} = \frac{\sum_i deg(g_{i,j})}{j}$ with $I_2(j)$. If $\frac{deg(g_j)}{j} < I_2(j)$ then there is a non-factor of degree j. For the primitive case we compare with $\frac{\phi(2^j-1)}{j}$.

If $\frac{deg(g_j)}{j} \geq I_2(j)$, the only way to determine whether all irreducible (primitive) polynomials of degree j are factors of g_j is to find those factors that appear in more than one of the $g_{i,j}$'s and to eliminate all their appearances except for one.

We considered two methods for removing repeated factors. The first is referred to as the *lcm method* and the second is referred to as the *gcd method.* The *lcm method* will be shown to be faster, but it also requires more space, which might not be available.

In the *lcm method* we first sort the $g_{i,j}$s according to their degrees and then place them in the sets s_k, where $g_{i,j} \in s_k$ iff $2^{k-1} < deg(g_{i,j}) \leq 2^k$. The sets $\{s_k\}$ are ordered according to their index, in increasing order. We then begin computing *lcms* of two polynomials taken from the first set. If this set has only one polynomial we take the second polynomial from the next set. The resulting *lcm* polynomial is placed in the set corresponding to its degree. This process ends when we are left with one polynomial, representing the *lcm* of all the polynomials $g_{i,j}$.

In the *gcd method* the polynomials $g_{i,j}$ are sorted by their degrees. In each iteration the polynomial with the highest degree is taken out of the set and and all pairwise *gcds* between itself and the other polynomials are taken. If the *gcd* is greater than 1, the other polynomial is divided by this *gcd*. At the end of the iteration none of the remaining polynomials in the set has a factor in common with the polynomial that was taken out. Thus, when the procedure ends, no factor appears in more than one of the $g_{i,j}$s.

Lemma 16:

1. The complexity of the second phase is $O(|H|^2 M(n) \log n)$.
2. The expected complexity of the second phase is $O(|H| \log^3 |H| L(n) \log(|H|n))$.

Proof:

1. We can bound the work required for the *lcm method* as follows. First assume $|H|$ and $d_{i,j}$ are powers of 2 (if they are not, for bounding purposes increase

them to the nearest power of 2). Also, assume the polynomials are leaves of a binary tree. All the polynomials in the same level have the same degree (each level corresponds to a different set s_k). Assume that in every *lcm* step, the degree of the *lcm* is the sum of the degrees of its two operands (i.e. the operands are relatively prime). The maximum degree the final *lcm* can have is $|H|n$ and computing this *lcm* costs $O(M(|H|n)\log(|H|n))$. Computing the two *lcm*'s of the next to last level costs at most $O(2\cdot M(|H|n/2)\cdot\log(|H|n/2))$. In each lower level there are at most twice as many *lcm*'s being computed but each costs less than half the cost of the level above it, hence the total cost is bounded by $O(\log(|H|n)M(|H|n)\log(|H|n)) \leq O(u^2M(|H|n))$.

To use the *lcm* method we need enough memory to store the final *lcm*. If we do not have the required memory, we use the *gcd method.* The work required is $O(|H|^2M(n)\log n)$.

2. When taking into account the expected size of the polynomials $g_{i,j}$, factorization becomes practical. The factoring algorithm used is that of Cantor and Zassenhaus [7]. The complexity for factoring a product of r distinct irreducible polynomials of the degree j is given by $O(rM(rj)(j+\log(rj))$ (see Appendix C). By Lemma 10, the expected number of polynomials $g_{i,j}$ that have more than $5\cdot 10^k$ factors is less than $|H|/10^{2k+2}$. If we take the number of polynomials with $5\cdot 10^k$ factors to be $\frac{99|H|}{10^{2k+2}}$ (i.e. all polynomials with at most 5 factors are assumed to have 5, all polynomials with 6...50 factors are assumed to have 50, etc.), then the expected work required to factor all the polynomials is bounded by

$$O\left(\sum_{k=0}^{\frac{1}{2}\log_{10}|H|-1}\frac{99|H|}{10^{2k+2}}5\cdot 10^k M(5j\cdot 10^k)(j+\log(5j\cdot 10^k))\right).$$

By using the fact that $5j\cdot 10^k < n$ and by writing $M(5j\cdot 10^k)$ as $5j\cdot 10^k\cdot L(n)$, we can bound the sum by

$$O\left(\sum_{k=0}^{\frac{1}{2}\log_{10}|H|-1} 25|H|jL(n)(j+\log n)\right) = O(|H|\log|H|jL(n)(j+\log n)). \quad (3.2)$$

When the factorization is completed, all the irreducible factors can be sorted in time $O(|H| \cdot \log |H|)$ and the unique factors can be counted.

Summing over $j = 1 \ldots u (= es(H))$ we get $O(|H| \log |H| u^2 L(n)(u + \log n))$. Since $u \approx \log |H|$, the expression becomes $O(|H| \log^3 |H| L(n) \log(|H|n))$.

$\square$

3.3.3 Finding a non-factor

We are now at the third phase, where we know the smallest degree d_a for which there exists a non-factor for h. We also have, $m \leq |H|$ polynomials g_{i,d_a} that are products of distinct irreducible (primitive) factors of h, all g_{i,d_a}'s are pairwise relatively prime and every irreducible (primitive) factor of degree d_a of h is a factor of one of these polynomials. We want to find an irreducible (primitive) polynomial of degree d_a that is a non-factor of H.

One approach is to divide the product of all irreducible (primitive) polynomials of degree d_a by the product of all m polynomials and find a factor of the result. This might pose a problem if we do not have the product at hand, i.e. only the polynomials g_{i,d_a}, or if the product is too large to handle as one polynomial.

Another way is to randomly select irreducible (primitive) polynomials and check whether they are factors or non-factors. The only way to check is by doing the actual division. This division, however, will be regular long division, and not FFT division, whenever the divisor has very small degree compared to the degree of the dividend. If an irreducible (primitive) polynomial is relatively prime to all of the g_{i,d_a}'s, it is a non-factor. If it divides at least one of the polynomials, we can keep the result of the division and reduce our work in upcoming trials. This reduction requires that polynomials do not repeat in the selection process.

Lemma 17:

1. The complexity of finding a non-factor once d_a is known is $O(|H|^2 n^2 d_a^2 M(d_a))$ for the irreducible case and $O(|H|^2 n^2 d_a^3 M(d_a) \log \log d_a)$ for the primitive case.

2. The expected complexity is $O(|H| \log |H| d_a M(d_a))$ for the irreducible case and $O(|H| \cdot \log |H| \cdot d_a^2 \log \log d_a M(d_a))$ for the primitive case.

Proof:

1. The procedure generates random polynomials, checks them for irreducibility (primitivity) and whether they are factors or not. The expected number of random polynomials that are tested for irreducibility (primitivity) before an irreducible (primitive) polynomial of degree d_a is found is $d_a/2$ ($\frac{d_a}{2} \log \log d_a$) [33]. The work required to test each polynomial for irreducibility is $O(d_a M(d_a))$ ($O(d_a^2 M(d_a))$) [33]. The sum of the $d_{i,j}$'s cannot exceed $|H|n$, therefore after at most $\frac{|H|n}{d_a}$ irreducible polynomials are tried, a non-factor is found. The work involved with each try is $|H|n \cdot d_a$ (long division). Thus, the expected work required to find a non-factor is $O(|H|^2 n^2 \cdot d_a^2 M(d_a))$. For the primitive case the work is $O(|H|^2 n^2 d_a^3 M(d_a) \log \log d_a)$. For a more detailed discussion see Appendix B.

2. If the polynomials $g_{i,j}$ were factored (see proof of Lemma 16,(2)), once d_a is known, we draw irreducible (primitive) polynomials until a non-factor is found. We expect no more than $|H|/d_a$ factors. When an irreducible (primitive) polynomial is drawn, it takes $O(\log |H|)$ to check whether it is a factor or not. Hence, the expected work required to find a non-factor, once d_a is known, is bounded by $O(|H| \log |H| d_a M(d_a))$ for the irreducible case and $O(|H| \log |H| d_a^2 M(d_a) \cdot \log \log d_a)$ for the primitive case.

□

3.4 Practical scenarios

In this section we discuss some practical scenarios for finding zero-aliasing polynomials. First, when we want a non-factor of a pre-specified degree. Second, when we want to find a non-factor fast. Third, we compare our algorithm for finding

a least degree non-factor with an exhaustive search over all irreducible (primitive) polynomials in ascending degrees. In some cases, this type of search will be faster.

3.4.1 Finding a non-factor of a pre-specified degree

In cases where the register is required to function as both a RA and a PG, a non-factor of a pre-specified degree is needed. Thus

Problem 3: Given a set of polynomials $H = \{h_1, h_2, \ldots, h_{|H|}\}$, with $deg(h_i) \leq n$, find an irreducible (primitive) non-factor of degree t for H. □

This problem is exactly the same as finding the least degree non-factor, except that we only need to consider the case of $j = t$, instead of iterating over all $1 \leq j \leq u$. We first compute the polynomials $g_{i,t}$, then determine whether a non-factor of degree t exists, and if so find one.

Lemma 18:

1. The complexity of finding a non-factor of degree t is $O(|H|^2 n^2 t^2 M(t))$ for the irreducible case and $O(|H|^2 n^2 t^3 M(t) \log\log t)$ for the primitive case.
2. The expected complexity is $O(|H| M(n)(t + \log n))$.

Proof:

1. Computing the polynomials $g_{i,t}$ involves computing $g_{i,t} = gcd(h_i, x^{2^t} - x)$ and for each $l \in md(t)$ computing $f_l = gcd(g_{i,t}, x^{2^l} - x)$ and $g_{i,t} = g_{i,t}/f_l$. The cost of the first gcd computation is $O(tM(d_i) + M(d_i)\log d_i)$. The cost of the $|md(t)|$ subsequent gcd and divisions is bounded by $O(\log t(tM(d_{i,t}) + M(d_{i,t})\log d_{i,t}))$. Substituting n for d_i and $d_{i,t}$ we get $O(\log t \cdot M(n)(t + \log n))$.

   Once we have the polynomials $g_{i,t}$, we need to sift out multiple instances of the same irreducible polynomial. When using the *gcd method*, this has a worst case cost of $O(|H|^2 M(n) \cdot \log n)$.

At this stage, we know whether a non-factor of degree t exists or not. If one exists, we carry out phase 3. This has a worst case complexity of $O(|H|^2n^2t^2 \cdot M(t))$. This is the dominant term for the whole process. The analysis is the same for the primitive case, hence the worst case complexity of finding an irreducible (primitive) non-factor of a given degree t for a set of polynomials H is $O(|H|^2n^2t^2M(t))$ $(O(|H|^2n^2t^3M(t)\log\log t))$.

2. We turn to analyze the expected complexity. For each h_i, we compute $g_{i,t} = gcd(h_i, x^{2^t} - x)$. This costs $O(|H|M(n)(t + \log n))$. The cost of sifting out the factors of degree less than t from the $g_{i,t}$'s, based on the expected number of factors for each degree, will be insignificant. Factoring and sorting the polynomials in the second phase has expected cost of $O(|H|\log|H|t \cdot L(n)(t + \log n))$ (Eq. (3.2)). The expected number of distinct irreducible factors of degree t of H is bounded by $|H|/t$. Thus, the cost of finding a non-factor at this stage which consists of drawing at most $\frac{|H|}{t}$ irreducible (primitive) polynomials, each at an expected cost of $O(\frac{t}{2}tM(t))$ $(O(\frac{t}{2}\log\log t \cdot t^2M(t)))$, and checking it against the list of factors, is bounded by $O(\frac{|H|}{t}\frac{t}{2}tM(t)\log(|H|/t))$ for the irreducible case and $O(|H|(\log|H| - \log t)t^2M(t)\log\log t)$ for the primitive case. Hence, the expected complexity of finding a non-factor of degree t for H is bounded by $O(|H|M(n)(t + \log n))$.

□

3.4.2 Finding a non-factor fast

Problem 4: Given a set of polynomials $H = \{h_1, h_2, \ldots, h_{|H|}\}$, with $deg(h_i) = d_i \leq n$, $\sum_{i=1}^{|H|} d_i = d_h$, find an irreducible (primitive) non-factor of H in less than 2^c tries. □

The sum of the degrees of all irreducible (primitive) polynomials of degree less than or equal to $s(d_h)$ $(p(d_h))$ is greater than d_h. If we look at $u = s\left(\frac{2^c}{2^c-1}d_h\right)$ $\left(u = p\left(\frac{2^c}{2^c-1}d_h\right)\right)$ then $\frac{\sum_{j=1}^{u} jI_2(j) - d_h}{\sum_{j=1}^{u} jI_2(j)} \geq 2^{-c}$ $\left(\frac{\sum_{j=1}^{u}\phi(2^j-1) - d_h}{\sum_{j=1}^{u}\phi(2^j-1)} \geq 2^{-c}\right)$ and if we draw

uniformly from all irreducible (primitive) polynomials of degree u, after 2^c drawings we expect to find a non-factor. The expected work cost for this case is $O(2^c \cdot (u^2M(u) + u|H|n)) = O(2^c u|H|n)$ which is the cost of 2^c iterations of drawing a polynomial and testing for irreducibility, and once one is found dividing all $|H|$ polynomials by this candidate non-factor, using long division. For the primitive case this becomes $O(2^c \cdot (u^3M(u)\log\log u + u|H|n)) = O(2^c u|H|n)$.

Example 3: Using the numbers in Example 1 again, say we want to find a non-factor in no more than 8 tries. We compute the bound $p\left(\frac{8}{7}10^{10}\right)$ and draw from all the primitive polynomials up to the computed bound. If we use Table 3.1, we see that instead of looking at the polynomials of degree less than or equal to 33, we need to consider all primitive polynomials of degree up to 34. In general, $\frac{2^c}{2^c-1} \leq 2$, hence by Lemma 7, we only have to consider polynomials of degree greater by at most 2 than for the case when we want the minimum degree non-factor. □

We can also use the expected bounds $es(d)$ and $ep(d)$ to lower the degrees of the candidate non-factors.

3.4.3 Exhaustive search

In this subsection, we compare our algorithms with an exhaustive search for a least degree non-factor. We will look at the irreducible case.

Assume the least degree irreducible non-factor has degree d_a. Also, assume we have a list of all irreducible polynomials in ascending order. The number of irreducible roots of degree j is less than 2^j. We can bound the work required to find the non-factor, by an exhaustive search, by $O(|H|n2^{d_a+1})$. Using the expected bound on d_a ($d_a = O(\log|H|)$), we can bound the work by $O(|H|^2n)$. The expected work required to find the least degree non-factor, by our algorithms, is $O(|H|u^2M(n))$, which becomes $O(|H|\log^2|H| \cdot n\log n)$ when we substitute in the value of u. Not taking into account any of the constants involved with these two results, their ratio is

$$\frac{|H|}{\log^2|H|\log n}.$$

Assuming $|H| = 1024$, this ratio is less than 1 for $n > 1210$. Assuming $|H| = 2048$, the ratio is less than 1 for $n > 124,500$. For $|H| = 4096$, the ratio is less than 1 for $n > 365,284,284$. This suggests that depending on the number of target faults and the length of the test sequence, an exhaustive search might be more effective. Assuming the number of faults is less than 4096, based on the expected bound on the degree of a non-factor, we would need to store all the irreducible polynomials of degree at most 13. The number of these polynomials is 1389.

3.5 Experimental results

The following experiments were conducted to verify our results. The experiments were conducted on a HP-700 workstation.

3.5.1 Random selections based on the absolute bounds

We generated a set of 1000 random polynomials of degree at most 200,000. The degree of the product of these polynomials was less than or equal to 200,000,000. We wanted a probability greater than 1/2 of finding a non-factor with just one drawing of a primitive polynomial. By looking at Table 3.1, we can achieve this by selecting from the set of all primitive polynomials of degree less than or equal to 29. In fact, the probability of success with the first drawing is greater than 3/4. We drew the polynomials in a 2 step process. The first step selected the degree of the primitive candidate, the second selected the candidate. In the first step we selected a 32 bit number and took its value modulo the number of primitive roots in the fields $GF[2]$ through $GF[2^{29}]$. The result was used to determine the degree of the primitive candidate, by looking at the first field $GF[2^d]$ such that the number of primitive roots in the fields $GF[2]$ through $GF[2^d]$ is greater than the result. The selection of the actual polynomial was done by setting the coefficients of $x, x^2, \ldots, x^{d-1}$ by a LFSR with a primitive feedback polynomial of degree $d-1$ that was initialized to a random state. This guarantees that no candidate will be selected twice and all candidates will have a chance at being considered. The candidates were tested for

primitivity and if they were, they were tested for being non-factors. If at some point they would turn out to be factors, the search continued from the current state of the degree $d-1$ LFSR.

We generated 200 different sets of 1000 polynomials, and for each set we searched for a non-factor. In *all 200 experiments* the first primitive candidate turned out to be a non-factor. Of the non-factors that were found, 1 was of degree 21, 2 were of degree 22, 3 of degree 23, 2 of degree 24, 7 of degree 25, 13 of degree 26, 32 of degree 27, 35 of degree 28 and 105 were of degree 29.

The number of polynomials that were tested for primitivity before one was found ranged from 1 to 160. The average number was 16. The time it took to find a primitive polynomial ranged from 0.01 seconds to 0.79 seconds. The average time was 0.104 seconds. It took between 153.25 and 166.68 seconds to find a non-factor, with the average being 160.50 seconds.

3.5.2 Random selections based on the expected bounds

Based on our expected bounds, Corollary 12, we should be able to find a non-factor of degree at most 14. We ran 100 experiment as above, only this time, we selected only primitive polynomials of degree 11 (the expected bound based on Table 3.1). The first primitive candidate that was selected was a non-factor in 66 of the 100 experiments. 19 experiments found the non-factor with the second candidate, 11 with the third, 2 with the fourth, 1 with the fifth and 1 with the sixth. We ran 100 experiments selecting only primitive candidates of degree 9. The number of primitive candidates that were tried before a non-factor was found ranged from 1 to 28. The average number of candidates was 7.5.

To test the tightness of our expected bound, we ran 126 experiments in which 1024 random polynomials of degree at most 200,000 were generated and an exhaustive search, in increasing order of degrees, was conducted to find the least degree non-factor. By our expected bound (Corollary 12), this least degree should be at most 14 and by Table 3.1 at most 11. In one experiment, the least degree was 7. In 35 it was 8 and in the remaining 90 experiments, the least degree was 9.

3.5.3 Experiments on benchmark circuits

We tried our worst case and expected bounds on error sequences of two circuits from the Berkeley synthesis benchmarks. The first circuit was *in5*, the second was *in7*. We used a fault simulator that did not take into account any fault collapsing, hence the number of faults was twice the number of lines in the circuit (for *stuck-at-0* and *stuck-at-1* faults on each line).

For circuit *in5* there were 1092 faults, six of which were redundant, hence there were 1086 detectable faults. The circuit had 14 primary outputs. We used a test sequence of length 6530 that detects all the non-redundant faults and computed the *effective output polynomials* of all the faults. All were non-zero, hence there were no cancellation of errors from one output by errors of another output. Thus we had 1086 error polynomials of degree at most 6543. From Table 3.1, the worst case bound on the degree of a primitive non-factor is 23. To draw a primitive non-factor with probability greater than $\frac{1}{2}$ we need to consider all primitive polynomials of degree 24 or less. We conducted 20 experiments of drawing zero-aliasing primitive polynomials, based on our worst case bounds. In all experiments, the first candidate was a non-factor. We then conducted another 20 experiments, this time drawing primitive polynomials of degree 14, the size of the register available at the circuit outputs. In all experiments the first candidate was a non-factor. Based on our expected bounds (Table 3.1), we should find a non-factor of degree 11 or less. We tried finding non-factor of degree 11, 9 and 7. In 17 of 20 degree 11 experiments, the first primitive candidate was a non-factor. Two experiments found the non-factor with the second try, one with the third. We conducted 15 degree 9 experiments before considering all 48 primitive polynomials of degree 9. Of the 48 primitive polynomials of degree 9, 33 were factors, and 15 were non-factors. The average number of candidates tried before a non-factor was found was $3\frac{1}{3}$. All 18 primitive polynomials of degree 7 were factors.

For circuit *in7* there were 568 faults, 567 of which were non-redundant. The circuit has 10 primary outputs and we used a test sequence of length 9280. Using the worst case bounds, to ensure selection of a primitive non-factor with probability greater than $\frac{1}{2}$, we considered all primitive polynomials of degree 24 or less. All 20

experiments found a non-factor with the first candidate. The expected bound (Table 3.1) for the degree of a primitive non-factor was 10. We tried to find non-factors of degree 11 and 10 (the size of the registers available at the outputs). All 20 degree 11 experiments found a non-factor with the first try. Of the 20 degree 10 experiments, 13 found a non-factor with the first try, 6 with the second and one with the third.

For both circuits we tried to find the least degree non-factor using an exhaustive search. Since the fault extractor we used did not do any fault collapsing, some of the error polynomials were identical. By summing the values of all non-zero erroneous output words for each simulated fault, we found at least 292 different error polynomials in *in7* and at least 566 different error polynomials in *in5*. This would make our expected bounds (Table 3.1) to be 9 for *in7* and 10 for *in5*. For both circuits the least degree non-factor had degree 8. It took 11 CPU minutes to find each of these polynomials.

3.6 Conclusions

In this paper we presented procedures for selecting zero-aliasing feedback polynomials for MISR-based RAs. When both PGs and RAs are designed as LFSRs/MISRs, our scheme, combined with algorithms for selecting efficient feedback polynomials for pattern generation (Chapter 2), enables the selection of one feedback polynomial that serves both tasks, thus reducing the overhead of reconfigurable registers.

We presented upper bounds on the least degree irreducible and primitive zero-aliasing polynomial for a set of modeled faults. We showed that in all practical test applications such a polynomial will always be of degree less than 53. In fact, by our expected bounds, when the number of faults is less than 10^6, this degree will be at most 21. In the experiments that were conducted, a zero-aliasing polynomial of degree less than the expected bound was always found.

We also presented procedures for finding a zero-aliasing polynomial, when the objective is to minimize the degree, to have a specific degree or speed. We analyzed the computational effort that is required both under worst case conditions

worst case		
	irreducible	primitive
bounds (u)	$s(d_h) = \lceil \log(\|H\|n+1) \rceil$	$p(d_h) = s(d_h) + \lceil 1 + \log\log(2s(d_h)) \rceil$
smallest non-factor	$\|H\|^2 n^2 u^2 M(u)$	$\|H\|^2 n^2 u^3 M(u) \log\log u$
degree t non-factor	$\|H\|^2 n^2 t^2 M(t)$	$\|H\|^2 n^2 t^3 M(t) \log\log t$
expected		
	irreducible	primitive
bounds (u)	$es(H) = \lceil \log \|H\| \rceil$	$ep(H) = 1 + es(H) + \lceil \log\log(2es(H)) \rceil$
smallest non-factor	$\|H\| M(n) u^2$	$\|H\| M(n) u^2$
degree t non-factor	$\|H\| M(n)(t + \log n)$	$\|H\| M(n)(t + \log n)$

Table 3.3: Summary of Results

and expected conditions. A (partial) summary of the results is presented in Table 3.3. For both the worst case analysis and expected analysis, Table 3.3 shows the upper bounds on the smallest non-factor, the computational complexity of finding a smallest non-factor and the complexity of finding a factor of a given degree.

If one is willing to look at two signatures, a great deal of computation effort can be saved due to the fact that most of the faults are detected early in the test sequence. For those faults, after the first signature we have low degree error polynomials which can be processed fast. Only the remaining faults need the (potentially) high degree error polynomials which take more time to process. A dynamic programming algorithm for finding the optimum placement of k signatures is given by Lambidonis et al. [23].

Based on our analysis and on our experiments, it is our conclusion that when the error polynomials of the modeled target faults are available, zero-aliasing is an *easily* achievable goal. Thus, to ensure high quality tests, a premium should be put on fault modeling, automated test pattern generator design and fault simulation. With these tools available, zero-aliasing is not a problem.

Reference List

[1] M.E. Aboulhamid and E. Cerny, *A Class of Test Generators for Built-In Testing*, IEEE Trans. Computers, Vol. C-32, No. 10, October 1983, pp. 957-959

[2] V.K. Agarwal and E. Cerny, *Store and Generate Built-In Testing Approach*, Proc. Intl. Sym. on Fault-Tolerant Computing, pp. 35-40, June 1981

[3] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974

[4] F. Berglez, C. Gloster and G. Kedem, *Built-In Self-Test with Weighted Random Pattern Hardware*, Proc. Intl. Conf. on Computer Design, pp. 161-166, 1990

[5] E.R. Berlekamp, *Factoring Polynomials Over Large Finite Fields*, Mathematics of Computation, Vol. 24, No. 111, pp. 713-735, July, 1970

[6] R.K. Brayton, G.D. Hachtel, C. McMullen and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, MA, 1984

[7] D.G. Cantor and H. Zassenhaus, *A New Algorithm for Factoring Polynomials over Finite Fields*, Mathematics of Computation, Vol. 36, No. 154, pp. 587-592, April, 1981

[8] K. Chakrabarty and J.P. Hayes, *Aliasing-Free Error Detection (ALFRED)*, Proc. 11th IEEE VLSI Test Sym., pp. 260-266, 1993

[9] D. Coppersmith, *Fast Evaluation of Logarithms in Fields of Characteristic Two*, IEEE Trans. Inform. Theory, Vol. IT-30, No. 4, pp. 587-594, July 1984

[10] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press, McGraw-Hill, 1990

[11] W. Daehn and J. Mucha, *Hardware Test Pattern Generation for Built-In Testing*, Proc. Intl. Test Conf., pp. 110-113, 1981

[12] R. Dandapani, J.H. Patel and J.A. Abraham, *Design of Test Pattern Generators for Built-In Test*, Proc. Intl. Test Conf., pp. 315-319, 1984

[13] C. Dufaza and G. Cambon, *LFSR based Deterministic and Pseudo Random Test Pattern Generator Structures*, Proc. 2nd European Test Conf., pp. 27-34, 1991

[14] G. Edirisooriya and J.P. Robinson, *A New Built-In Self-Test Method Based on Prestored Testing*, Proc. VLSI Test Sym., pp. 10-16, 1993

[15] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, 1979.

[16] R.L. Graham, D.E. Knuth and O. Patashnik, *Concrete Mathematics*, Addison-Wesley Publishing Company, 1989

[17] S.K. Gupta, D.K. Pradhan and S.M. Reddy, *Zero Aliasing Compression*, Proc. 20th Int. Sym. on Fault-Tolerant Computing, pp. 254-263, 1990

[18] D. Ha, *Automatic Test Pattern Generators and Fault Simulators for Combinational and Sequential Circuits*, Technical Report, Virginia Polytechnic, June 1992

[19] D.R. Heath-Brown, *Zero-free regions for Dirichlet L-functions, and the least prime in an arithmetic progression*, Proc. London Math. Soc., Vol. 64, No. 3, pp. 265-338, 1992

[20] S. Hellebrand, S. Tarnick and J. Rajski, *Generation of Vector Patterns Through Reseeding of Multiple-Polynomial Linear Feedback Shift Registers*, Proc. Intl. Test Conf., pp. 120-129, 1992

[21] K. Ireland and M. Rosen, *A Classical Introduction to Modern Number Theory*, Springer-Verlag New York Inc., 1982

[22] T. Kameda, S. Pilarski and A. Ivanov, *Notes on Multiple Input Signature Analysis*, IEEE Trans. Computer, Vol. 42, No. 2, pp. 228-234, February 1993

[23] D. Lambidonis, V.K. Agarwal, A. Ivanov and D. Xavier, *Computation of Exact Fault Coverage for Compact Testing Schemes*, Proc. Intl. Sym. Computers and Systems, pp. 1873-1876, 1991

[24] A. Lempel, G. Seroussi and S. Winograd, *Complexity of multiplication in finite fields*, Theoretical Computer Science, Vol. 22, pp. 285-296, 1983

[25] R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, Cambridge University Press, 1986

[26] A. Majumdar and S. Sastry, *On the Distribution of Fault Coverage and Test Length in Random Testing of Combinational Circuits*, Proc. Design Automation Conf., pp. 341-346, 1992

[27] F. Muradali, V.K. Agarwal and B. Nadeau-Dostie, *A New Procedure for Weighted Random Built-In Self-Test*, Proc. Intl. Test Conf., pp. 660-669, 1990

[28] A.M. Odlyzko, *Discrete Logarithms in Finite Fields and Their Cryptographic Significance*, Adv. in Cryptology (Proc. of Eurocrypt '84), Lecture Notes in Computer Science, Vol. 209, Springer-Verlag, New York, pp. 224-314, 1984

[29] C. Pomerance, *A Note on the Least Prime in an Arithmetic Progression*, J. Number Theory, Vol. 12, pp.218-223, 1980

[30] I. Pomeranz, S.M. Reddy and R. Tangirala, *On Achieving Zero Aliasing for Modeled Faults*, Proc. European Design Automation Conf., 1992

[31] D.K. Pradhan, S.K. Gupta and M.G. Karpovsky, *Aliasing Probability for Multiple Input Signature Analyzer*, IEEE Trans. Computer, Vol. 39, No. 4, pp. 586-591, April 1990

[32] S.C Pohlig and M. Hellman, *An Improved Algorithm for Computing Logarithms over $GF[p]$ and Its Cryptographic Significance*, IEEE Trans. Inform. Theory, Vol. IT-24, No. 1, pp. 106-110, January 1978

[33] M.O. Rabin, *Probabilistic Algorithms in Finite Fields*, SIAM J. of Computing, Vol. 9, No. 2, pp. 273-280, May 1980

[34] I.S. Reed and T.K. Truong, *The Use of Finite Fields to Compute Convolutions*, IEEE Trans. Inf. Th., Vol. IT-21, No. 2, pp. 208-213, March 1975

[35] P. Ribenboim, *The Book of Prime Number Records*, 2nd Edition, Springer-Verlag, 1989

[36] R.M. Robinson, *The Converse of Fermat's Theorem*, Amer. Math. Monthly, Vol. 64, pp. 703-710, 1957

[37] R.M. Robinson, *A Report on Primes of the Form $k2^n + 1$ and on Factors of Fermat Numbers*, Proc. Amer. Math. Soc., Vol. 9, pp. 673-680, 1958

[38] N.R. Saxena, P. France and E.J. McCluskey, *Simple Bounds on Serial Signature Aliasing for Random Testing*, IEEE Trans. Computer, Vol. 41, No. 5, pp. 638-645, May 1992

[39] J. Savir and P.H. Bardell, *On Random Pattern Test Length*, IEEE Trans. Computers, Vol. 33, No. 6, pp. 467-474, June 1984

[40] J. Savir and P.H. Bardell, *Built-In Self-Test: Milestones and Challenges*, VLSI Design, Vol. 1, No. 1, pp. 23-44, 1993

[41] A. Schönhage, *Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2*, Acta Informatica, Vol. 7, 1977, pp. 395-398

[42] M. Serra, T. Slater, D.M. Miller and J.C. Munzio, *The Analysis of One-Dimensional Cellular Automata and their Aliasing Properties*, IEEE Trans. CAD, Vol. 9, No. 7, pp. 767-778, July 1990

[43] C.W. Starke, *Built-In Test for CMOS Circuits*, Proc. Intl. Test Conf., pp. 309-314, 1984

[44] B. Vasudevan, D.E. Ross, M. Gala and K.L. Watson, *LFSR Based Deterministic Hardware for At-Speed BIST*, Proc. VLSI Test Sym., pp. 201-207, 1993

[45] K.D. Wagner, C.K. Chin and E.J. McCluskey, *Pseudorandom Testing*, IEEE Trans. Computers, Vol. 36, No. 3, pp. 332-343, March 1987

[46] J.A. Waicukauski, E. Lindenbloom, E.B. Eichelberger and O.P. Forlenza, *A Method For Generating Weighted Random Patterns*, IBM J. Res. Develop., pp. 149-161, March 1989

[47] T.W. Williams, W. Daehn, M. Gruetzner and C.W. Starke, *Aliasing Errors in Signature Analysis Registers*, IEEE Design and Test, pp. 39-45, April 1987

[48] D. Xavier, R.C. Aitken, A. Ivanov and V.K. Agarwal, *Using an Asymmetric Error Model to Study Aliasing in Signature Analysis Registers*, IEEE Trans. Computer-Aided Design, Vol. 11, No. 1, pp. 16-25, January 1992

Appendix A

The least prime in an arithmetic progression

Given an integer n, we would like to know what is the smallest prime of the form

$$p = k2^n + 1. \tag{A.1}$$

Alternatively, what is the smallest k such that p is a prime.

In Subsection A.1 we review some of the known upper bounds on the least prime in a general arithmetic progression. In Subsection A.2 we show additional evidence that, for most cases, k is not too large. In Subsection A.3, assuming k is not too large, we review known methods to find p and a primitive element in $GF[p]$. Finally, in Table A.1, for $0 \leq n \leq 206$, we show the least $1 \leq k \leq 100$, if one exists, such that $k2^n + 1$ is a prime.

A.1 General Bounds

The question of what is the smallest prime of the form of Eq. A.1 has received a lot of attention under the topic of *the smallest prime in an arithmetic progression.* We review some known facts, as given in [35, Ch. 4, Sec. IV.B].

For $d \geq 2$, $a \geq 1$, relatively prime, let $p(d, a)$ be the smallest prime in the arithmetic progression $\{a + kd \mid k \geq 0\}$. Can one find an upper bound, depending

only on a and d for $p(d,a)$? Let $p(d) = max\{p(d,a) \mid 1 \leq a < d, gcd(a,d) = 1\}$. Can one find an upper bound on $p(d)$ depending only on d?

In 1950, Erdös showed

For every $d \geq 2$ and for every $C > 0$, there exists $C' > 0$ (depending on C), such that

$$\frac{\#\{a \mid 1 \leq a < d\ ,\ gcd(a,d) = 1\ ,\ p(d,a) < C\phi(d)\log d\}}{\phi(d)} \geq C'.$$

While this states that for every d there are values of a that satisfy the inequality for $p(d,a)$, it says nothing about $a = 1$.

In 1971, Elliot and Halberstam showed the following.

For every $\epsilon > 0$, for every $d \geq 2$, not belonging to a set of density 0

$$\#\{1 \leq a < d\ ,\ gcd(a,d) = 1\ ,\ p(d,a) < \phi(d)(\log d)^{1+\epsilon}\} = \phi(d) - o(\phi(d))$$

The above bound is not valid for every a and excludes some of the d's.

In 1977, Kumar Murty established that

$$p(d) < d^{2+\epsilon}$$

for every $\epsilon > 0$ and d not belonging to a sequence of density 0.

Again, the bound may not be valid for some powers of 2.

In 1987, Bombieri, Friedlander and Iwaniec proved a theorem from which the following estimate for $p(d,a)$ can be deduced

$$p(d,a) < d^2/(\log d)^k$$

for every $k > 0$, $d \geq 2$, $gcd(a,d) = 1$ and d is outside a set of density 0.

With the generalized Reimann hypothesis, Heath-Brown showed that

$$p(d) \leq C(\phi(d))^2(\log d)^4.$$

In 1944, Linnik proved the following theorem.

There exists $L > 1$ such that $p(d) < d^L$ for every sufficiently large $d \geq 2$.

A lot of effort has gone into estimating L, called *Linnik's constant.* The best result is that of Heath-Brown which showed that Linnik's theorem holds unconditionally for $L = 5.5$ [19].

Heath-Brown (1978) conjectured that $p(d) \leq Cd(\log d)^2$ and Wagstaff (1979), based on heuristic arguments, showed that $p(d) \sim \phi(d)(\log d)^2$.

While there is plenty of proof that there's a very good chance of finding a small prime given (A.1), the only guarantee on its size is given by Heath-Brown's estimation of Linnik's constant, and this, too, is only for sufficiently large powers of 2. Most researchers believe the Heath-Brown conjecture.

A.2 Some evidence for the likelihood of $k < 2^n$

Let q be a prime. Looking at the values of $p = k2^n + 1$, p will be divisible by q iff

$$k2^n \equiv -1 \pmod{q}.$$

Since q and 2^n are relatively prime, the first k for which q divides p is

$$k_q \equiv -2^{-n} \pmod{q} \quad , \quad k_q < q.$$

After that, q will divide p for $k \equiv k_q \pmod{q}$.

Let $Q(n)$ be the set of primes q such that $2 < q < 2^n$. We would like to know whether there exists a $1 \leq k \leq 2^n$ that will set (A.1) to be prime. Since p will be

less than $2^{2n}+2$, if for any k, p is composite, it is divisible by one of the primes in $Q(n)$. Hence, numbering the elements of $Q(n)$ as $q_1, \ldots, q_{|Q(n)|}$, p is prime iff

$$p \not\equiv k_{q_i} \pmod{q_i} \;\; \forall i \in \{1 \ldots |Q(n)|\}. \tag{A.2}$$

If we look at all the numbers in the interval $[1 \ldots 2^n]$, and mark those that are congruent to $k_{q_i} \pmod{q_i}$ for all $q_i \in Q(n)$, then the first unmarked number (if one exists) is the least k that will produce a prime number. For example, if $n = 3$ then $Q(n) = \{3, 5, 7\}$ and $k_3 = 1$, $k_5 = 3$ and $k_7 = 6$. The marked numbers are 1,3,4,6 and 7 and for $k = 2$ we have $p = 17$, a prime. We would like to show that an unmarked number always exists. This is dependent on the starting positions of the k_{q_i}'s. The k_{q_i}'s can not take on any value, only values that are additive inverses of values in the multiplicative group of 2 modulo q_i. We will look at a broader question, and assume that the k_{q_i}'s can take on any value.

Let $Q = \prod_{i=1}^{|Q(n)|} q_i$. Mark off all the numbers in the interval $I = [0 \ldots Q]$ that are multiples of any of the q_i's. By the *Chinese Remainder Theorem*, any non zero number in I has a unique representation modulo all the q_i's, and in particular one such number, r, can be represented as $(q_1 - k_{q_1}, \ldots, q_{|Q(n)|} - k_{q_{|Q(n)|}})$. In the interval $[r \ldots r + 2^n]$ the marked numbers are of the form $k_i + lq_i$, hence when substituting an unmarked numbers for k, we get a number in the form of (A.2).

An integer i is said to be $Q(n)$-*anti-smooth* if it has no prime factors in $Q(n)$. We would like to show that in I, every sub-interval $[i \ldots i + 2^n]$ has at least one $Q(n)$-*anti-smooth* number. In particular, this would imply that the interval $[r \ldots r + 2^n]$ has at least one $Q(n)$-*anti-smooth* number and, hence, the least k that would set (A.1) to a prime is less than or equal to 2^n.

A broader question, which would imply this result, was conjectured to be true [35, pp. 219-220] [29]. Unfortunately, it was proven by Pomerance [29] to be false. For our case, this means that there exists some N, such that for $n > N$, there is at least one interval with no $Q(n)$-anti-smooth numbers. This does not rule out the possibility that k is less than 2^n even for those large n's, because the *"bad"* interval is not necessarily our interval. In fact, we will show that the average distance between

unmarked numbers is less than n, implying that not only is it probable that our interval contains an unmarked number, but that one is likely to be found very fast, hence k is small.

Lemma 19: The marked off numbers in I are symmetric about $Q/2$.

Proof: For all i, 0 and Q are marked, q_i and $Q - q_i$ are marked, and for all $0 \leq j \leq Q/q_i$, $j \cdot q_i$ and $Q - j \cdot q_i$ are marked. □

The number of $Q(n)$-anti-smooth numbers in the interval $[1 \ldots Q]$ is the number of integers that are relatively prime to Q. This number is given by *Euler's function*

$$\phi(n) = n \prod_p (1 - \frac{1}{p})$$

for all distinct prime divisors p of n. Rosser and Schoenfeld proved the following lower bound on $\phi(n)$ [35, p. 172]

$$\phi(n) \geq e^{-\gamma} \frac{n}{\ln \ln n + \frac{5}{2e^{\gamma} \ln \ln n}} \quad \text{for all } \; n \geq 3 \tag{A.3}$$

$\gamma = 0.577215665\ldots$ [35, p. 162] is *Euler's constant.*

The only exception to the bound is for $n = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$, for which $\frac{5}{2}$ is replaced by 2.50637.

Using the lower bound for $\phi(n)$ on Q, we find the average difference between $Q(n)$-anti-smooth number to be

$$\begin{aligned}
\frac{Q}{\phi(Q)} &\leq \frac{Q}{e^{-\gamma} \frac{Q}{\ln \ln Q + \frac{5}{2e^{\gamma} \ln \ln Q}}} \\
&= \frac{2e^{\gamma} (\ln \ln Q)^2 + 5}{2e^{-\gamma} e^{\gamma} \ln \ln Q} \\
&= e^{\gamma} \ln \ln Q + \frac{5}{2 \ln \ln Q}
\end{aligned}$$

$$< \ln\ln Q \qquad \text{for} \qquad Q > e^{2^3} = e^8. \tag{A.4}$$

Since $Q \approx e^{2^n}$ ($\lim_{2^n \to \infty} Q = e^{2^n}$, [21, p.27]), the average difference between unmarked numbers is less than n.

A.3 Finding the Least Prime in the Arithmetic Progression $k2^n + 1$

The procedure for finding the least prime in the arithmetic progression $k2^n + 1$ is basically a sequential search over k. The bounds on the size of the smallest prime provide "good" evidence that we will not have to search for long. Once we find the prime, we also need to find a generator of the multiplicative group.

The basic step is a primality test due to Proth (1878) [35, p. 40] [36, Th. 9]

Theorem 20: Let $N = k2^n + 1$, where $0 < k < 2^n$. Suppose that $(a/N) = -1$. Then N is prime iff

$$a^{(N-1)/2} \equiv -1 \pmod{N}$$

(a/N) is the *Jacobi symbol* [35, pp. 35-36]. □

A corollary to Theorem 20 is given by Robinson [36].

Corollary 21: Let $N = k2^n + 1$, where $n > 1, 0 < k < 2^n$, and $3 \nmid k$. Then N is prime iff

$$3^{(N-1)/2} \equiv -1 \pmod{N}$$

□

Thus, in most cases, we can choose $a = 3$ in Theorem 20. When $3|k$, we need another value for a. If N is not a perfect square (if it is, then it's not a prime) then the probability that the Jacobi symbol of a random choice for a will be -1 is $1/2$, hence we can easily find a value for a that will satisfy the hypothesis of Theorem 20.

Once we've found a prime N, we need to find a generator for its multiplicative group. If α is a generator, than so is α^j for all $(j, N-1) = 1$, hence the probability of a random choice being a generator is $\frac{\phi(N-1)}{N-1}$. Assuming $k < 2^n$, then by (A.4), the probability is greater than $\frac{1}{\ln n + \frac{1}{3}}$ for $n \geq 6$. In testing whether an element is a generator, we'll need the factorization of $N-1$. Again, assuming $N < 2^{2n}$, we can afford the work.

In constructing addition and multiplication tables modulo p, it is advantageous to choose the smallest generator. Assuming k is small and the generators are uniformly distributed, we can test the elements sequentially until we find the smallest generator.

In [37], Robinson found all the primes of the form $k2^n+1$ with k odd, $1 < k < 100$ and $0 < n < 512$. From the tables of [37], we constructed Table A.1, which for $0 \leq n \leq 206$ gives the least $k \leq 100$, if one exists, such that $k2^n + 1$ is a prime. While Robinson considered only odd k's, we wanted the smallest k, be it odd or even. From Table A.1, we see that for $1 \leq n \leq 100$, there are only 8 cases for which $k > n$. For 6 of those $k < 2n$. For the other 2, for $n = 100$ k is at most 232 and for $n = 99$ k is at most 464. For $101 \leq n \leq 206$, there are 26 cases where we do not know the least k. For 8 of those we know it is less than n, and for another 8 we know it is less than $2n$. For the remaining 10 values of n, we can only say that k is (much) less that $17n$.

Table A.1: Smallest $1 \leq k \leq 100$ for which $k2^n + 1$ is prime, $0 \leq n \leq 206$

n	k	n	k	n	k	n	k	n	k	n	k	n	k	n	k	n	k
0	1	23	20	46	19	69	23	92	7	115	*	138	*	161	18	184	25
1	1	24	10	47	27	70	39	93	29	116	*	139	*	162	9	185	29
2	1	25	5	48	15	71	39	94	43	117	56	140	61	163	75	186	24
3	2	26	7	49	14	72	40	95	77	118	28	141	*	164	93	187	12
4	1	27	15	50	7	73	20	96	57	119	14	142	58	165	74	188	6
5	3	28	12	51	14	74	10	97	86	120	7	143	29	166	37	189	3
6	3	29	6	52	7	75	5	98	43	121	81	144	43	167	30	190	7
7	2	30	3	53	20	76	93	99	*	122	93	145	68	168	15	191	*
8	1	31	35	54	10	77	29	100	*	123	42	146	34	169	*	192	*
9	15	32	18	55	5	78	15	101	*	124	21	147	17	170	93	193	*
10	12	33	9	56	27	79	36	102	58	125	11	148	81	171	56	194	*
11	6	34	12	57	29	80	18	103	29	126	10	149	*	172	28	195	*
12	3	35	6	58	54	81	9	104	43	127	5	150	*	173	14	196	96
13	5	36	3	59	27	82	13	105	53	128	21	151	65	174	7	197	48
14	4	37	15	60	31	83	20	106	37	129	21	152	81	175	27	198	24
15	2	38	10	61	36	84	10	107	*	130	*	153	63	176	*	199	12
16	1	39	5	62	18	85	5	108	93	131	72	154	*	177	56	200	6
17	6	40	6	63	9	86	87	109	74	132	36	155	*	178	28	201	3
18	3	41	3	64	12	87	75	110	37	133	18	156	*	179	14	202	*
19	11	42	9	65	6	88	61	111	30	134	9	157	83	180	7	203	72
20	7	43	9	66	3	89	51	112	15	135	*	158	43	181	83	204	36
21	11	44	15	67	9	90	28	113	*	136	*	159	72	182	100	205	18
22	25	45	31	68	31	91	14	114	*	137	*	160	36	183	50	206	9

Appendix B

Generating irreducible polynomials

There are two ways to randomly select irreducible polynomials. The first is by randomly selecting polynomials and checking whether they are irreducible. The second is by randomly selecting elements in $GF[2^d]$, verifying they are not in any of it's subfields and constructing their minimal polynomial.

The first method is discussed by Rabin [33]. An irreducible polynomial of degree d has $d+1$ coefficients, some of which are 0. Both the most significant and least significant coefficients are non-zero, in our case "1". The number of non-zero coefficients must be odd, otherwise the polynomial is divisible by $(x+1)$. Thus, we have 2^{d-2} possible choices for the remaining coefficients. Call a particular assignment f.

Lemma 22: [33, Lemma 1,p. 275]
Let $l_1, \ldots, l_k$ be all the unique prime divisors of d. Let $m_i = d/l_i$ and let the set $md(d) = \{m_i\}_{i=1}^{k}$. A polynomial $f \in GF[2][x]$ of degree d is irreducible in $GF[2][x]$ iff

$$\begin{array}{llcll} (1) & f & \mid & x^{2^d} - x & \\ (2) & gcd(f, x^{2^u} - x) & = & 1 & , \text{ for all } u \in md(d) \end{array}$$

□

The work required for Step (1) is one reduction modulo f. This costs $O(dM(d))$ operations. Step (2) requires k gcd operations. Notice that while reducing $(x^{2^d} - x)$

modulo f (Section 3.2.2) the intermediate results include the values of $(x^{2^{m_i}} - x)$ modulo f, hence these reductions need not be computed again. The cost, is then, $O(kM(d)\log d)$. Since $k < \log d$ and for $d \geq 16$ we have $d \geq \log^2 d$, the overall complexity for checking whether a degree d polynomial is irreducible is $O(dM(d))$.

For the primitive case, the test is [25, Ch. 3]

$$\begin{array}{lrcll} (1) & f & \mid & x^{2^d-1}-1 & \\ (2) & gcd(f, x^v - 1) & = & 1 & , \quad \text{for all } v \in md(2^d-1). \end{array}$$

There is more work involved in this case. First, we must compute $x^{2^j} \bmod f \equiv r_j$ for $1 \leq j < d$ and then multiply and reduce the appropriate r_j's to get the equivalent of x^v. Computing the r_j's can be done in $O(dM(d))$. Denoting $|md(2^d-1)|$ by q, computing the x^v's will cost $O(qdM(d)) \leq O(d^2M(d))$. The gcd operations will add another $O(qM(d)\log d)$. Thus the overall work is $O(d^2M(d))$.

By Lemma 5

$$\frac{2^{d-1}}{d} < I_2(d) \leq \frac{2^d}{d}$$

thus, the probability, pr_{ir}, of choosing an irreducible polynomial is

$$\frac{2}{d} < pr_{ir} = \frac{I_2(d)}{2^{d-2}} \leq \frac{4}{d}$$

and the expected number of tries, before an irreducible polynomial is found, is less than $d/2$.

For the primitive case, we have, by Lemma 4, for $d > 6$

$$\frac{2^d}{d \cdot 2.08 \cdot \log\log d} < \frac{\phi(2^d-1)}{d} < \frac{2^d}{d}$$

thus, the probability, pr_{pr}, of choosing a primitive polynomial is

$$\frac{4}{d \cdot 2.08 \cdot \log\log d} < pr_{pr} = \frac{\phi(2^d-1)}{2^{d-2}d} \leq \frac{4}{d}$$

and the expected number of tries, before a primitive is found, is less than $(d \cdot 2.08 \cdot \log\log d)/4$.

If care is taken not to generate the same f more than once, there is no need to keep track of the irreducible polynomials tested.

The second method for selecting irreducible polynomials, is by selecting elements of $GF[2^d]$, verifying they are not elements of any subfield of $GF[2^d]$ and constructing the corresponding minimal polynomial. We assume we are given a primitive polynomial, g, that defines the arithmetic of $GF[2^d]$ and a primitive element, α, a root of g. We randomly select an element by randomly selecting a power of α.

Two construction methods for minimal polynomials are discussed in [25, p. 94] and [33]. The first involves reducing α^s mod g, $O(dM(d))$, constructing the matrix whose j-th row is α^{sj} mod g, for $0 \leq j \leq d$, $O(dM(d))$ and finding a set of dependent rows which includes rows 0 and d, $O(d^3)$. The major work load is in this last step, which will dominate the primitive case as well.

The second construction is based on the fact that

$$m_{\alpha^s} = \prod_{i=0}^{d-1}(x - \alpha^{s2^i})$$

where m_{α^s} is the minimal polynomial of α^s. We first need to represent α^s in terms of a polynomial in α of degree $< d$, $O(dM(d))$. Next, we need to square and reduce this polynomial $d-1$ times, $O(dM(d))$. Finally, we need to perform the multiplication. We start from the left, multiplying the accumulated result by the next linear factor. The result is

$$\left(\sum_{i=0}^{j} a_i x^i\right)(x + \alpha^{s2^{j+1}}) = a_j x^{j+1} + \sum_{i=1}^{j}(a_{i-1} + a_i \alpha^{s2^{j+1}})x^i + a_0 \alpha^{s2^{j+1}}.$$

Each multiplication costs $M(d)$, each addition costs d. There are $d(d-1)/2+(d-1)$ multiplications and $(d)(d-1)/2$ additions, for a total cost of $O(d^2 M(d))$.

In addition to the higher cost, over the case of randomly selecting polynomials, we must keep track of the generated polynomials, even if powers are generated just once, since every irreducible polynomial has d roots, thus a chance of being generated up to d times.

Appendix C

Factoring a product of distinct irreducible polynomials of the same degree

We consider two methods for factoring a product of distinct irreducible polynomials of the same degree. The first is due to Rabin [33], which is a generalization of the method proposed by Berlekamp [5]. Consider the Trace polynomial [25, pp. 50-59] over $GF[2^d]$

$$Tr(x) = x + x^2 + \cdots + x^{2^{d-1}}.$$

$Tr(x)$ is a linear function from $GF[2^d]$ to $GF[2]$. 2^{d-1} elements of $GF[2^d]$ are sent to 0 and the other 2^{d-1} elements are sent to 1. Let f be a polynomial of degree $n = dt$ which we wish factor. Consider $f_1 = gcd(f, Tr(x))$. If not all the roots of f are sent to 0 or 1 by the trace function, then f_1 is a non-trivial factor of f.

Theorem 23: [33, Theorem 6]
Let δ be an element of $GF[2^d]$. Let $\alpha_1 \neq \alpha_2$ be elements of $GF[2^d]$. Then

$$2^{d-1} = |\{\delta : Tr(\delta\alpha_1) \neq Tr(\delta\alpha_2)\}|.$$

□

Corollary 24: [33, Corollary 5]
For $\delta \in GF[2^d]$, consider $f_\delta = gcd(f_1, Tr(\delta x))$. We have

$$1/2 \leq prob(\delta : 0 < deg(f_\delta) < deg(f_1))$$

Proof: The roots of f_δ are those α_i which are roots of f_1 and $Tr(\delta x)$. But for α_1, α_2 roots of f_1, with probability 1/2, $Tr(\delta\alpha_1) \neq Tr(\delta\alpha_2)$. □

Rabin comments that the actual probability is nearly $1-1/2^k$, where $k = deg(f_1)$, but he can not prove this.

The root finding algorithm is to compute f_1, select δ randomly, and compute f_δ. If $deg(f_\delta) \leq \frac{1}{2}deg(f_1)$, set $f_2 = f_\delta$, otherwise $f_2 = f_1/f_\delta$. In at most $\log n$ iterations a root is found and all that is left is to construct its minimal polynomial.

The problem we have with this algorithm is that it involves use of the Trace function, thus arithmetic in $GF[2^d]$ instead of $GF[2]$ and the construction of minimal polynomials. Although this can be dealt with, it would complicate the required programs.

The second method we consider is that of Cantor and Zassenhaus [7]. We first review the method for factoring polynomials over $GF[q]$ for odd q, followed by Cantor and Zassenhaus' modification when q is a power of 2, and in particular when $q = 2$. We follow this review with our analysis of the algorithm's complexity for factoring a product of distinct irreducible polynomials of the same degree. We then show that when d, the degree of the irreducible factors, and q are even, then the case where $q \equiv 2 \bmod 3$ can be treated in the same manner as the case where $q \equiv 1 \bmod 3$.

Let $f = \prod_{i=1}^{r} u_i$ be a polynomial over $GF[q]$, q odd, with the u_i being distinct irreducible polynomials of degree d.

By *Euclid's algorithm*, there exist g_i, c_i such that

$$g_i u_i + c_i \prod_{j \neq i} u_j = 1. \tag{C.1}$$

Since $c_i \equiv (\prod_{j\neq i} u_j)^{-1} \bmod u_i$, we can assume $deg(c_i) < d$.

Set $e_i = c_i \prod_{j \neq i} u_j$, then

$$e_i \equiv \begin{cases} 1 \bmod u_i \\ 0 \bmod u_j \quad , \quad j \neq i \end{cases}$$

with $deg(e_i) < deg(f)$.

Let b_i, $1 \leq i \leq r$, be a set of polynomials of degree $< d$. By the *Chinese Remainder Theorem (CRT)* [3, Ch. 8], there exists a unique polynomial b, $deg(b) < rd$, s.t. for all i

$$b \equiv b_i \bmod u_i.$$

Furthermore,

$$b = \sum_{i}^{r} (b_i e_i \bmod f).$$

Suppose we have a polynomial

$$a(x) = \sum_{i=1}^{r} a_i e_i$$

with $a_i \in \{0, \pm 1\}$ and suppose $a(x) \neq 0, \pm 1$.

Let α be a root of f, hence α is a root of u_k, for some k. Therefore,

$$a(\alpha) = \sum_{i=1}^{r} a_i e_i(\alpha) = a_k e_k(\alpha).$$

By (C.1), we have

$$1 = g_k(\alpha) u_k(\alpha) + e_k(\alpha) = e_k(\alpha)$$

which yields

$$a(\alpha) = a_k$$

hence $a(\alpha) - a_k = 0$ and u_k divides $a(x) - a_k$. Therefore, let $S = \{i : a_i = 0\}$, $T = \{i : a_i = 1\}$, $R = \{i : a_i = -1\}$. Then,

$$\begin{aligned} gcd(f, a(x) - 1) &= \prod_{i \in T} u_i \\ gcd(f, a(x)) &= \prod_{i \in S} u_i \end{aligned}$$

$$gcd(f, a(x)+1) \quad = \quad \prod_{i \in R} u_i$$

At least one of the above equations will result in a non-trivial factorization of f. The question is how to find such a polynomial a.

A polynomial a is found by a random search. Choose a random non-constant polynomial b from $GF[q][x]$, with all such polynomials given the same probability, $1/(q^n - q)$. By the *CRT*, b can be written as

$$b = \sum_{i=1}^{r} (b_i e_i \bmod f)$$

with $b_i \equiv b \bmod u_i$ (note we do not know the u_i's).

Set $m = (q^d - 1)/2$.

$$b^m \equiv \left(\sum_{i=1}^{r} b_i^m e_i \right) \bmod f$$

with b_i^m equal to $0, 1$ or -1. Thus, $b^m \bmod f$ is of the form of the polynomial $a(x)$ that we are looking for. It remains to be seen what is the probability of $b^m \bmod f$ turning out to be ± 1 (since b was chosen to be non-constant, it was not 0, hence $b^m \not\equiv 0$).

For each u_i, there are m b_i's such that $b_i^m \equiv 1 \bmod u_i$ and m b_i's such that $b_i^m \equiv -1 \bmod u_i$.
For b^m to equal $\pm 1 \bmod f$, we need all b_i^m to equal $1 \bmod u_i$ or all b_i^m to equal $-1 \bmod u_i$. Thus, there are $2m^r - (q-1)$ non-constant polynomials b that, when raised to the $m-th$ power, will equal $\pm 1 \bmod f$. There are $q^n - q$ possible b's, hence the probability that $b^m \equiv \pm 1 \bmod f$ is

$$\frac{2m^r - q + 1}{q^n - q} = \frac{2^{1-r}(q^d - 1)^r - q + 1}{q^n - q} < 2^{1-r} \leq 1/2$$

giving a probability of success exceeding $1 - 2^{1-r} \geq 1/2$.

Turning to the case where q is a power of 2, if $q \equiv 1 \bmod 3$, then $GF[q]$ contains a $3rd$ root of unity ρ. By setting $m = (q^d - 1)/3$ and choosing b as before, define

$$a \equiv b^m \equiv \sum_{i=1}^{r} a_i e_i \bmod f$$

where $a_i \in GF[4]$. If $a \notin GF[4]$, it is guaranteed that one of $gcd(f, a - y)$, $y \in GF[4]$, results in a non-trivial factor of f. The probability that a turns out to be in $GF[4]$ is

$$\frac{3m^r - q + 1}{q^n - q} < 3^{1-r} \leq 1/3.$$

When $q \equiv 2 \bmod 3$ (as is the case when $q = 2$), Cantor and Zassenhaus suggest to factor f in the quadratic extension $GF[q^2]$ of $GF[q]$ and combine conjugate factors over $GF[q]$. Conjugate factors can be matched in the following way. For the case $q = 2$, if we denote a root of an irreducible polynomial of degree d by α, then the remaining $d - 1$ roots are α^{2^j} for $1 \leq j \leq d - 1$. Over $GF[4]$, this irreducible polynomial factors into two polynomials of degree $\frac{d}{2}$, the roots of one being α^{2^j} for even j's and the roots of the other being α^{2^j} for odd j's. Denote these polynomials as f' and f''. Since α is a root of f', $f'(\alpha) = 0$. Therefore $f'(\alpha)^2 = (\sum_{i=0}^{d/2} f'_i \alpha^i)^2 = 0$. But over fields of characteristic 2 we have

$$(\sum_{i=0}^{d/2} f'_i \alpha^i)^2 = \sum_{i=0}^{d/2} (f'_i)^2 (\alpha^i)^2 = \sum_{i=0}^{d/2} (f'_i)^2 (\alpha^2)^i = 0$$

hence

$$f''(x) = \sum_{i=0}^{d/2} (f'_i)^2 x^i.$$

The complexity of the algorithm is as follows. The polynomial b is raised to the power m and reduced modulo f. This can be done in $O(dM(n))$ operations. On average, b is chosen three times, thus the complexity remains the same. The gcd operations that follow can be done in $O(M(n) \log n)$ operations. There are r factors, hence this process repeats r times. The complexity, without combining factors, is thus, $O(rM(n)(d + \log n))$. Combining conjugate factors requires $O(2r \log(2r))$ to sort the factors and $O(r(d + \log(2r) + M(d)))$ to create the combined factors

by selecting a random factor, squaring its coefficients, finding its conjugate and multiplying the two. This is majorized by the factoring term.

We now show that when d is even, or when r/d is large enough, we can achieve a high probability of success, without working in the quadratic extension $GF[q^2]$ of $GF[q]$ (i.e. we can choose the coefficients of b to be from $GF[q]$).

We consider the case of $q = 2$, but the discussion remains similar when q is a power of 2. The polynomial b is chosen at random from the set of non-zero polynomials of degree $< dr = n$ over $GF[2]$, all given the same probability.

There are two cases to consider. The first is when d is odd, the second when d is even.

For the case when d is odd, the u_i's remain unchanged when working over $GF[4]$, i.e. they do not factor and the irreducible factors of f, over $GF[4]$, are exactly those over $GF[2]$.

Setting $m = (4^d - 1)/3$, we have $m = (2^d - 1)(2^d + 1)/3 = k(2^d - 1)$.

The b_i's can be seen as elements in $GF[2^d]$. Such an element, if not 0, when raised to the m-th power will be 1.

We will get a non-trivial factor only when not all the reduced values are 1, i.e. at least one $b_i^m \bmod u_i = 0$. This will happen only if $b_i \equiv 0 \bmod u_i$, making b divisible by u_i. So, the fraction of b's that will result in a non-trivial factorization is the fraction of b's that are divisible by at least one of the u_i's. By the *inclusion-exclusion* counting principle, the number of b's that are divisible by at least one u_i is

$$\binom{r}{1} 2^{n-d} - \binom{r}{2} 2^{n-2d} + \binom{r}{3} 2^{n-3d} - \binom{r}{4} 2^{n-4d} + \cdots + (-1)^{r-2} \binom{r}{r-1} 2^d$$

which is equal to

$$(2^d)^r - (2^d - 1)^r + (-1)^r \sim 2^{dr} \left(1 - \left(\frac{2^d - 1}{2^d}\right)^r\right).$$

The total number of b's is $2^{dr}-1$, hence our chance of finding a non-trivial factor is dependent on the values of r and d.

When d is even, the situation is much better. This time, each u_i factors into $u_{i,1}$ and $u_{i,2}$, each of degree $d/2$. Consequently, we have

$$m = \frac{4^{d/2}-1}{3} = \frac{2^d-1}{3}.$$

When working in the quadratic extension, the elements of $GF[4]$ are scalars, hence we can find a non-trivial factor by looking at $gcd(f, a-\gamma)$, $\gamma \in GF[4]$. Over $GF[2]$, the only scalars we can use are 0 and 1.

In order to get a non-trivial factor we first need $a \notin GF[4]$ and second we need at least one $b_i^m \bmod u_i$ to equal 0 or 1. Another way of looking at this is that to fail to get a non-trivial factor, either all $(b_i^m \bmod u_i)$ equal 1 or all $(b_i^m \bmod u_i) \in \{\delta, \delta^2\}$ for δ primitive in $GF[4]$. Thus

$$\begin{aligned} prob(failure) &= prob(\forall i : b_i^m \bmod u_i = 1) + \\ &\quad prob(\forall i : b_i^m \bmod u_i \in \{\delta, \delta^2\}) \\ &= \frac{m^r}{2^{dr}-1} + \frac{(2m)^r}{2^{dr}-1} \\ &= \frac{\left(\frac{2^d-1}{3}\right)^r}{2^{dr}-1} + \frac{\left(\frac{2(2^d-1)}{3}\right)^r}{2^{dr}-1} \\ &< \left(\frac{1}{3}\right)^r + \left(\frac{2}{3}\right)^r. \end{aligned}$$