Analysis and Construction of Galois Fields for Efficient Storage Reliability

Technical Report UCSC-SSRC-07-09

Kevin M. Greenan (kmgreen@cs.ucsc.edu)
Ethan L. Miller (elm@cs.ucsc.edu)
Thomas J. E. Schwarz, S.J.[†](tjschwarz@scu.edu)

Storage Systems Research Center
Jack Baskin School of Engineering
Computer Science Department
University of California, Santa Cruz
Santa Cruz, CA 95064
`http://ssrc.cse.ucsc.edu/`

[†]Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053

August 21, 2007

Abstract

Software-based Galois field implementations are used in the reliability and security components of many storage systems. Unfortunately, multiplication and division operations over Galois fields are expensive, compared to the addition. To accelerate multiplication and division, most software Galois field implementations use pre-computed look-up tables, accepting the memory overhead associated with optimizing these operations. However, the amount of available memory constrains the size of a Galois field and leads to inconsistent performance across architectures. This is especially problematic in environments with limited memory, such as sensor networks.

In this paper, we first analyze existing table-based implementation and optimization techniques for $GF(2^l)$ multiplication and division. Next, we propose the use of techniques that perform multiplication and division in an extension of $GF(2^l)$, where the actual multiplications and divisions are performed in a smaller field and combined. This approach allows different applications to share Galois field multiplication tables, regardless of the field size, while drastically lowering memory consumption. We evaluated multiple such approaches in terms of basic operation performance and memory consumption. We then evaluated different approaches for their suitability in common Galois field applications. Our experiments showed that the relative performance of each approach varies with processor architecture, and that CPU, memory limitations and field size must be considered when selecting an appropriate Galois field implementation. In particular, the use of extension fields is often faster and less memory-intensive than comparable approaches using standard algorithms for $GF(2^l)$.

1 Introduction

The use of Galois fields of the form $GF(2^l)$, called *binary extension fields*, is ubiquitous in a variety of areas ranging from cryptography to storage system reliability. These algebraic structures are used to compute codewords in linear erasure codes, evaluate and interpolate polynomials in Shamir's secret sharing algorithm [17], compute algebraic signatures over variable-length strings of symbols [16] and encrypt blocks of data in Rijndael's cipher. These applications typically perform computation in either $GF(2^8)$ or $GF(2^{16})$.

Multiplication in $GF(2^8)$ and $GF(2^{16})$ is typically done using pre-computed look-up tables, while addition of two elements in $GF(2^l)$ is usually, but not always, carried out using an inexpensive bitwise-XOR of the elements. As a result, overall algorithm speed is more affected by multiplication because, while it is more expensive than addition, most algorithms use it just as often. Thus, optimizing the multiplication operation will, in turn, lead to much more efficient applications of Galois fields.

The byte-based nature of computer memory motivates the use of $GF(2^8)$—each element represents one byte of storage. However, the small number of distinct elements in $GF(2^8)$ poses an advantage and a disadvantage. Multiplication in $GF(2^8)$ is typically carried out using a complete 64 KB multiplication table, which should fit into the L2 cache of most processors. Unfortunately, there are only 256 distinct field elements in $GF(2^8)$, restricting the order of any single element to at most 255. Thus, codewords in a Reed-Solomon code defined over $GF(2^8)$ are restricted to no more than 257 or 258 symbols [9].

The smallest feasible field larger than $GF(2^8)$ is $GF(2^{16})$, which allows Reed-Solomon to have $65,537$-symbol codewords. Unfortunately, growing to $GF(2^{16})$ and beyond has a significant impact on field operations such as multiplication. A complete lookup table for multiplication in $GF(2^{16})$ requires 8 GB—well beyond the memory capacity of most systems. The standard alternative to a full multiplication table is a logarithm and an antilogarithm table that requires 256 KB; this may fit in the standard L2 cache, but might be partially evicted from an L1 cache. Moreover, using table-based multiplication approaches in $GF(2^{32})$ and larger is simply impossible given modern memory sizes.

Composite fields are an alternative to table-based methods. Using this technique, elements of a field $GF(2^n)$ are represented in terms of elements in a sub-field $GF(2^l)$, where $n = l \times k$. The composite field $GF((2^l)^k)$ is a representation of a *k-degree extension* of $GF(2^l)$, where $GF(2^l)$ is called the *ground field*. Compared to traditional table-based methods, this technique trades additional computation for a significant decrease in storage space. Since the cost of a cache miss amounts to the execution of a few instructions, trading additional computation for less storage can significantly increase performance. Additionally, many applications will use Galois fields for different purposes; using the composite field technique, it might be possible to reuse the the ground Galois field tables for several variable-sized Galois field extensions.

Distinct Galois field implementation techniques will perform differently on disparate hardware due to the complex interactions between the various cache levels, speculative execution of instructions, and other factors. The techniques for Galois field implementation described here are not new, but, to our knowledge, there is no comparative study of their performance over standard, contemporary computing systems. We were motivated to evaluate the relative performance of our implementation techniques by the design of large storage systems that will use Galois fields to provide different functionalities such as integrity checking, erasure protection, and cryptographic protection.

The performance of the Galois field implementation may be a critical factor in overall system performance. The results presented here show dramatic differences in throughput, though (as expected) yield no overall algorithmic winner on the various platforms we studied. We conclude that a performance optimizing software suite needs to tune the Galois field implementations to the hardware. As the industry shifts to multi-core systems with shared memory, but private L1 caches, we expect a need to redo the performance evaluation. It would also be interesting to see how our various implementation alternatives perform on low-power systems used to build sensor networks. We restrict ourselves in this paper to efficient implementation of operations in $GF(2^4)$, $GF(2^8)$, $GF(2^{16})$, and $GF(2^{32})$, postponing an evaluation of implementations for larger size fields.

The contributions of this paper are threefold. First, we present and compare popular table-based binary extension field implementation techniques and their optimizations. Next, we propose the use of software-based composite fields when implementing $GF(2^{16})$ and $GF(2^{32})$. Finally, we compare all of the Galois field representations in terms of raw multiplication and application performance in three distinct environments.

We begin in Section 2 by describing a few well-known applications of Galois fields in computer systems. Section 3 presents some background on Galois fields and shows common multiplication techniques in binary extension fields and a few optimizations to these techniques. We describe the construction of a composite field in Section 4. Our implementation is described and evaluated in Section 5. We present related work in Section 6 and we conclude the paper in Section 7.

2 Applications of Galois fields

The implementation and examination of erasure codes in disk arrays, distributed storage systems and content distribution systems has been a common area of research within the systems community over the past few years. Most work is concerned with fault tolerant properties of codes, performance implications of codes, or both. Most of the erasure codes used in storage systems are XOR-based and generally provide limited levels of fault tolerance; a flood of special-purpose, XOR-based codes is the result of a performance-oriented push from the systems side [4, 2, 19]. While these codes perform all encoding and decoding using the XOR operator, they either lack flexibility in the number of tolerated failures or are not maximum distance separable (MDS) and may require additional program complexity.

Linear erasure codes, such as Reed-Solomon [12], are MDS. As a result, Reed-Solomon codes provide flexibility and optimal storage efficiency. Unfortunately, Reed-Solomon codes are generally regarded as inefficient because encoding and decoding require Galois field arithmetic. Some effort has gone into alternative representations of Reed-Solomon codes. Arithmetic over field elements $GF(2^l)$ may be transformed into operations in $GF(2)$, where multiplication is the bit-AND operation [3, 14]. While multiplication in a binary extension field is avoided, performance is heavily dependent on the choice of the code's generator matrix and the alternative representation results in additional program code complexity. Furthermore, the benefits of the XOR-based Reed-Solomon codes are generally effective when encoding large pieces of data. Compared to XOR-based Reed-Solomon and other special-purpose coding techniques, we believe that the use of Galois field arithmetic in linear codes leads to simple, generalized implementations.

Threshold cryptography algorithms, such as Shamir's secret sharing algorithm [17], also rely on Galois fields for encoding and decoding. A random k-degree polynomial over a Galois field is chosen, where the zeroth coefficient is a secret to be shared among n participants. The polynomial is evaluated over n coordinates (shares), distributed among the participants. Polynomial interpolation is used to reconstruct the zeroth coefficient from any $k+1$ unique shares. The construction, evaluation and interpolation of the polynomial may also be done over Z_p for some prime number p. Unfortunately, when dealing with large fields, the use of a suitable prime number may result in field elements that are not byte-aligned. Using Galois fields allows all of the field elements to be byte aligned.

Another class of algorithms that use Galois field arithmetic is algebraic signatures [16]. Algebraic signatures are *Rabinesque* because of the similarity between signature calculation and the hash function used in the Rabin-Karp string matching algorithm [5]. The algebraic signature of a string $s_0, s_1, \ldots, s_{n-1}$ is the sum $\sum_{i=0}^{n-1} s_i \alpha^i$, where α and the elements of the string are members of the same Galois field. Algebraic signatures are typically used across RAID stripes, where the signature of a parity disk equals the parity of the signatures of the data disks. This property makes the signatures well-suited for efficient, remote data verification and data integrity in distributed storage systems.

All of these applications make extensive use of Galois field multiplication, which is generally second to disk access as a performance bottleneck in a storage system that uses Galois fields. We describe methods aimed at improving general multiplication performance in the next two sections.

3 Construction of $GF(2^l)$

The field $GF(2^l)$ is defined by a set of 2^l unique elements that is closed under both addition and multiplication, in which every non-zero element has a multiplicative inverse and every element has an additive inverse. Addition and multiplication in a Galois field are associative, distributive and commutative. The Galois field $GF(2^l)$ may be represented by the set of all polynomials of degree at most $l-1$, with coefficients from the binary field $GF(2)$—the field defined over the set of elements 0 and 1. Thus, the 4-bit field element $a = 0111$ has the polynomial representation $a(x) = x^2 + x + 1$.

In contrast to finite fields defined over an integer prime, the field $GF(2^l)$ is defined over an irreducible polynomial of degree l with coefficients in $GF(2)$. An irreducible polynomial is analogous to a prime number in that it cannot be factored into two non-trivial factors. Addition and subtraction in $GF(2)$ is done with the bitwise XOR operator, and multiplication is the bitwise AND operator. It follows that addition and subtraction in $GF(2^l)$ are also carried out using the bitwise XOR operator, while multiplication turns out to be more complicated. In order to multiply two elements $a(x), b(x) \in GF(2^l)$, we perform polynomial multiplication of $a(x) \cdot b(x)$ and reduce the product modulo an

```
**Input :** $a, b \in GF(2^l)$ and $FLD\_SIZE = 2^l$
**if** $a$ is 0 || $b$ is 0 **then**
    **return** 0
**end if**
$sum \leftarrow log[a] + log[b]$
**if** $sum \geq FLD\_SIZE - 1$ **then**
    $sum \leftarrow sum - FLD\_SIZE - 1$
**end if**
**return** $antilog[sum]$
```

Figure 1: Computing the product of a and b using log and antilog tables

l-degree irreducible polynomial over $GF(2)$. Division among field elements is computed in a similar fashion using polynomial division.

The *order* of a non-zero field element $\mathrm{ord}(\alpha)$ is the smallest positive i such that $\alpha^i = 1$. If the order of an element $\alpha \in GF(2^l)$ is $2^l - 1$, then α is *primitive*. In this case, α generates $GF(2^l)$, *i. e.*, all non-zero elements of $GF(2^l)$ are powers of α. For a detailed and rigorous explanation of finite fields, please refer to [8].

The goal of this section is to show several approaches to performing multiplication over the fields $GF(2^4)$, $GF(2^8)$, and $GF(2^{16})$ and to show a few optimizations. All of the methods described here may be used to perform ground field calculations in the composite field representation.

3.1 Multiplication in a $GF(2^l)$

Courses in Algebra often define Galois fields as a set of polynomials over a prime field such as $\{0, 1\}$ modulo a generator polynomial. While it is possible to calculate in Galois fields performing polynomial multiplication and reduction modulo the generator polynomial, doing so is rarely efficient. Instead, we can make extensive use of pre-computed lookup tables. For small Galois fields, it is possible to calculate all possible products between the field elements and store the result in a (full) look-up table. However, this method consumes large amounts of memory.

Log/antilog tables make up for storage inefficiency by requiring some computation and extra lookups in addition to the single lookup required for a multiplication table. The method is based on the existence of a primitive element α. Every non-zero field element $\beta \in GF(2^l)$ is a power $\beta = \alpha^i$ where the *logarithm* is uniquely determined modulo $2^l - 1$. We write $i = \log(\beta)$ and $\beta = \mathrm{antilog}(i)$. The product of two non-zero elements $a, b \in GF(2^l)$ can be computed as $a \cdot b = \mathrm{antilog}(\log(a) + \log(b)) \mod 2^l - 1$. The algorithm for computing the product of two field elements using logarithm and antilogarithm tables is shown in Figure 1.

3.2 Optimization of the Full Multiplication Table

We can shrink the full multiplication table by breaking a multiplier in $GF(2^l)$ into a left and a right part. We store the result of multiplication by a left and by a right part in two tables, resulting in a significantly smaller multiplication table. Multiplication is performed using a lookup into both tables and an addition to calculate the correct product. Other papers have called this optimization a "double table" [16]; we call it a *left-right table*. To define it formally, we represent the elements of $GF(2^l)$ as polynomials of degree up to $l - 1$ over $\{0, 1\}$. If we wish to multiply field elements $a(x) = a_1 + a_2x + \cdots + a_{l-1}x^{l-1}$ and $b(x) = b_1 + b_2x + \cdots + b_{l-1}x^{l-1}$ in $GF(2^l)$, the product $a(x) \cdot b(x)$ can be arranged into two products and a sum

$$\begin{aligned} & (a_1 + \cdots + a_{l-1}x^{l-1}) \cdot (b_1 + \cdots + b_{l-1}x^{l-1}) \\ = \; & ((a_1 + \cdots + a_{\frac{l}{2}-1}x^{\frac{l}{2}-1}) \cdot (b_1 + \cdots + b_{l-1}x^{l-1})) \\ + \; & ((a_{\frac{l}{2}}x^{\frac{l}{2}} + \cdots + a_{l-1}x^{l-1}) \cdot (b_1 + \cdots + b_{l-1}x^{l-1})). \end{aligned}$$

By breaking the result into two separate products, we can construct two tables having $2^{\frac{l}{2}} \cdot 2^l$ entries each, assuming l is even. This approach, which computes the product of two elements using two lookups, a bitwise shift, two bitwise ANDs and a bitwise XOR, requires that tables be generated as shown in Figure 2. The product $a \cdot b,\ a, b \in GF(2^l)$ is the sum of $mult\_tbl\_left[a_1 >> \frac{l}{2}][b]$ and $mult\_tbl\_right[a_0][b]$, where a_1 are the $\frac{l}{2}$ most significant bits and a_0 are the $\frac{l}{2}$ least significant bits.

```
**Input :** $FLD\_SIZE == 2^l$
**for** $i = 0$ to $FLD\_SIZE \gg \frac{l}{2}$ **do**
    **for** $j = 0$ to $FLD\_SIZE$ **do**
        $mult\_tbl\_left[i][j] \leftarrow gf\_mult(i \ll \frac{l}{2}, j)$
        $mult\_tbl\_right[i][j] \leftarrow gf\_mult(i, j)$
    **end for**
**end for**
```

Figure 2: Precomputing products for the left and right multiplication tables

While this multiplication table optimization may seem minor for smaller fields, it is highly effective for $GF(2^8)$ and $GF(2^{16})$. The standard multiplication table for $GF(2^8)$ requires 64 KB, which should fit in the L2 cache, but might not fit in the data section of an L1 cache. Using the multiplication table optimization, the table for $GF(2^8)$ is 8 KB and has a better chance of fitting in the L1 cache. The table for $GF(2^{16})$ occupies 8 GB and will not fit in main memory in a majority of systems; however, the optimization shrinks the table from 8 GB to 66 MB, which has a much better chance of fitting into main memory.

3.3 Optimization of the Logarithm/Antilogarithm Method

While our previous optimization traded an additional calculation for space savings, the optimizations for the log/antilog go in the opposite direction. As shown in Figure 1, the standard log/antilog multiplication algorithm requires two checks for zero, three table-lookups and an addition modulo $2^l - 1$. We can divide two numbers by taking the antilogarithm of the difference between the two logarithms, but this difference has to be taken modulo $2^l - 1$, too. We can avoid the cumbersome reduction modulo $2^l - 1$ by observing that the logarithm is defined to be a number between 0 and $2^l - 2$, so the sum of two logarithms can be at most $2 \cdot 2^l - 4$ and the difference between two logarithms is larger or equal to $-2^l + 1$. By extending our antilogarithm table to indices between $-2^l + 1$ and $2^l - 2$, our log/antilog multiplication implementation has replaced the addition/subtraction modulo $2^l - 1$ with a normal signed integer addition/subtraction.

If division is a rare operation, we can use an insight by A. Broder [personal communication from Mark Manasse] to speed up multiplication by avoiding the check for the factors being zero. Although the logarithm of zero is undefined, if the logarithm of zero is defined to be a negative number small enough that any addition with a true logarithm still yields a negative number, no explicit zero check is needed. Thus, we set $\log(0) = -2^l$ and then define the antilog of a negative number to be 0. As a result of this redefinition, the antilog table now has to accommodate indices between -2^{l+1} and $2^{l+1} - 2$ and has about quintupled in size, but the product of a and b may now be calculated simply as $a \cdot b = \text{antilog}[\log[a] + \log[b]]$.

Huang and Xu proposed three improvements to the log/antilog approach [7]. The improvements were compared to the full multiplication table and the unoptimized logarithm/antilogarithm approaches in $GF(2^8)$. The first two improvements optimize the modular reduction operation out and maintain the conditional check for zero, while the third improvement is Broder's scheme. The first improvement replaces the modulus operator by computing the product of two non-zero field elements as

$$\text{antilog}[(\log[a] + \log[b])\&(2^n - 1) + (\log[a] + \log[b]) \gg n]$$

The second improvement extends the indices of the antilogarithm table to account for the sum of two logarithms exceeding $2^n - 1$, where the product of two non-zero field elements is computed the same way it is computed in Broder's scheme. Due to the similarity between the second and third improvements in Huang and Xu [7] and Broder's scheme, we chose to only use the first improvement (called Huang and Xu) for comparison in our study.

We have covered a variety of techniques and optimizations for multiplication in $GF(2^l)$. Figure 3 lists the space and computation requirements for each multiplication scheme. In the next section, we show that the multiplication techniques in $GF(2^l)$ can be used to efficiently compute products over the field $GF((2^l)^k)$.

4 Using Composite Fields

Many hardware implementations of Galois fields use the composite field technique [10, 11], in which multiplication in a large Galois field is implemented in terms of a smaller, intermediate Galois field. Specializing to field sizes that

	Space	Complexity
Mult. Table	$2^l \cdot 2^l$	1 LKU
Lg/Antilg	$2^l + 2^l$	3 LKU, 2 BR, 1 MOD 1 ADD
Lg/Antilg Optimized	$5 \cdot 2^l$	3 LKU, 1 ADD
Huang and Xu	$2^l + 2^l$	3 LKU, 1 BR, 3 ADD 1 SHIFT, 1 AND
LR Mult. Table	$2^{(\frac{3}{2}l)+1}$	2 LKU, 2 AND 1 XOR, 1 SHFT

Figure 3: Ground field memory requirements and computation complexity of multiplication in $GF(2^l)$. The operations are abbreviated LKU for table lookup, BR for branch and MOD for modulus; the rest refer to addition and the corresponding bitwise operations.

are a power of 2, we want to implement multiplication and division in $GF(2^n)$ in terms of $GF(2^l)$, where l divides n and $n = l \cdot k$. We know from Galois field theory that $GF(2^n)$ is isomorphic to an extension field of $GF(2^l)$ generated by an irreducible polynomial $f(x)$, (*i. e.* one without non-trivial dividers) of degree k with coefficients in $GF(2^l)$.

In this implementation, elements of $GF(2^n)$, written $GF((2^l)^k)$, are polynomials of degree up to $k-1$ with coefficients in $GF(2^l)$. In our standard representation, each element of $GF((2^l)^k)$ is a bit string of length n, which we now break into k consecutive strings of length l each. For example, if $n = 32$ and $k = 4$, a bit string of length 32 is broken into four pieces of length 8. If the result is (a_3, a_2, a_1, a_0), we identify the $GF(2^{32})$ element with the polynomial $a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0$, with coefficients $a_3, a_2, a_1, a_0 \in GF(2^8)$. This particular representation is denoted $GF((2^8)^4)$.

The product of two $GF((2^l)^k)$ elements is obtained by multiplying the corresponding polynomials—let them be $a_{k-1} \cdot x^{k-1} + \ldots + a_2 \cdot x^2 + a_1 \cdot x + a_0$ and $b_{k-1} \cdot x^{k-1} + \ldots + b_2 \cdot x^2 + b_1 \cdot x + b_0$—and reducing the result modulo the irreducible, defining polynomial $f(x)$. We multiply the two polynomials out to obtain

$$\sum_{i=0}^{2(k-1)} \left(\sum_{\nu+\mu=i} a_\nu \cdot b_\mu \right) x^i$$

For $i > k-1$ in this expression, we replace x^i with $x^i \mod f(x)$, multiply out, and reorganize according to powers of x^i. The result is the product in terms of products of the coefficients of the two factor polynomials multiplied with coefficients of the defining polynomial $f(x)$. In order to do this efficiently, we must search for irreducible polynomials $f(x)$ of degree k over $GF(2^l)$ that have many coefficients equal to zero or to one.

For small field sizes, it is possible to exhaustively search for irreducible polynomials. If $k \leq 3$, an irreducible polynomial is one that has no root and irreducibility testing is simple. Otherwise, the Ben-Or algorithm [6] is an efficient way to find irreducible polynomials.

We have implemented $GF((2^l)^2)$, for $l \in \{4, 8, 16\}$ and $GF((2^l)^4)$, for $l \in \{4, 8\}$ using the composite field representation. The remainder of this section discusses multiplication and inversion in the composite field representation.

4.1 Multiplication in $GF((2^l)^2)$

In general, irreducible polynomials over $GF(2^n)$ of degree two must have a linear coefficient. We have found irreducible polynomials of the form $f(x) = x^2 + s \cdot x + 1$ over $GF(2^4)$, $GF(2^8)$, and $GF(2^{16})$ that will do well for our purpose. We write any element of $GF((2^l)^2)$ as a linear or constant polynomial over $GF(2^l)$. Multiplying $a_1 \cdot x + a_0$ by $b_1 \cdot x + b_0$ gives the product

$$\begin{aligned} & (a_1 \cdot x + a_0) \cdot (b_1 \cdot x + b_0) \\ = \quad & a_1 b_1 x^2 + (a_1 b_0 + a_0 b_1) x + a_0 b_0. \end{aligned}$$

Since $x^2 = sx + 1 \mod f(x)$, this becomes

$$(a_1 b_0 + a_0 b_1 + s a_1 b_1) x + (a_1 b_1 + a_0 b_0).$$

As described above, multiplication in $GF((2^l)^2)$ is done in terms of five multiplications in $GF(2^l)$, of which one is done with a constant element s. Parenthetically, if we define $GF(2^8)$ through the binary polynomial $x^8+x^4+x^3+x^2+1$, or `0x11D` in hexadecimal notation, we can choose s to be `0x3F` in hexadecimal notation, resulting in an irreducible polynomial that is indeed optimal in the number of resulting multiplications. An irreducible, quadratic polynomial must have three non-zero coefficients since, without a constant coefficient the polynomial has root zero and a polynomial of form x^2+a always has the square root of a as a root. Since x^2+x+1 is not irreducible over $GF(2^4)$, $GF(2^8)$ or $GF(2^{16})$, we can do no better than $x^2+s\cdot x+1$. The fields $GF((2^4)^2)$, $GF((2^8)^2)$ and $GF((2^{16})^2)$ were implemented using this representation.

4.2 Multiplication in $GF((2^l)^4)$

We can use the composite field technique in two ways to implement $GF((2^l)^4)$. First, we can implement $GF(2^8)$ and $GF(2^{16})$ as $GF((2^4)^2)$ and $GF((2^8)^2)$, respectively, and then implement $GF(2^{32})$ as $GF(((2^8)^2)^2)$. This approach would require that we find an irreducible polynomial over $GF((2^8)^2)$, but it turns out that there is one of the same form as in the previous section. Mutatis mutandis, our multiplication formula remains valid and we have implemented multiplication in $GF(2^{32})$ in terms of $5\cdot 5=25$ multiplications in $GF(2^8)$. The same approach applies to multiplication in $GF(2^{16})$ over coefficients in $GF((2^4)^2)$.

We can also use a single step in order to implement $GF(2^{32})$ in terms of $GF(2^8)$, but finding an appropriate irreducible polynomial of degree 4 in $GF(2^8)$ is more involved. After exhaustive searching, we determined that we can do no better than with x^4+x^2+sx+t, for which the resulting implementation uses only 22 multiplications, 16 of which result from multiplying all coefficients with each other and the remaining 6 from multiplying s and t by a_3b_3, $a_3b_2+a_2b_3$, and by $a_3b_1+a_2b_2+a_1b_3$, reusing some of the results. For instance, we have an addend of $a_3b_3(t+1)x^2$ and of a_3b_3t, but we can calculate both of them with a single multiplication by t. Again, the same formula works for the $GF((2^4)^4)$ representation of $GF(2^{16})$.

4.3 Multiplicative Inverse in Composite Fields

Division among field elements is required when interpolating a polynomial or solving a system of linear equations. This makes division essential for decoding in Reed-Solomon and Shamir's secret sharing. Given a small field $GF(2^l)$, division is efficiently accomplished by determining a multiplicative inverse using logarithm and antilogarithm tables. Unfortunately, when working in a large field, there is no room for a table that covers all of the field elements so multiplicative inverses must be computed on-the-fly. In general, the Extended Euclidean Algorithm [5] is the fastest way to compute multiplicative inverses in a composite field. We employ the Extended Euclidean Algorithm for extensions of $GF(2^l)$ of degree larger than 2. When working in the field $GF((2^l)^2)$ we take advantage of the ground field log/antilog tables to compute the multiplicative inverse.

Given two elements $a,b\in GF((2^l)^2)$, we know that $a\cdot b=1\Leftrightarrow b=a^{-1}$. With respect to an irreducible polynomial $x^2+sx+1\in GF((2^l)^2)$, we can determine the multiplicative inverse of some $a\in GF((2^l)^2)$ by solving for b_0 and b_1 in

$$(a_0b_1+a_1b_0+(s)a_1b_1)x+(a_0b_0+a_1b_1)=1 \tag{1}$$

Solving Equation 1 for b_0 and b_1, the inverse of $a=a_1x+a_0$ may be computed with 5 multiplications, 3 inversions and 4 additions in $GF(2^l)$. We could perform a similar boot-strapped calculation for 4-degree extensions, but chose to limit the inversion optimization to 2-degree extensions.

There are three cases for finding the multiplicative inverse b of a. If $a_0=0$, then by setting $b_0=sa_1^{-1}$ and $b_1=a^{-1}$ we get $a_1a_1^{-1}+(a_1(sa_1^{-1})+sa_1(a_1^{-1})\alpha=1\in GF((2^l)^2)$, which results in $b=sa_1^{-1}+a_1\alpha$. If $a_1=0$, then we set $b_0=a_0^{-1}$ and $b_1=0$ to get $b=a_0^{-1}+0\alpha$. Finally, if $a_0\neq 0$ and $a_1\neq 0$, we can solve Equation 1 for $c\in GF(2^l)$, where $c=a_1b_1$ and $c+1=a_0b_0$. Since substituting c into $a_0b_0+a_1b_1=1$, we need to solve

$$\begin{aligned}
& a_0b_1+a_1b_0+(s)a_1b_1=0\\
\Leftrightarrow\quad & (c)a_0a_1^{-1}+(c+1)a_1a_0^{-1}+(s)(c)a_1a_1^{-1}=0\\
\Leftrightarrow\quad & (a_1a_0^{-1})(a_1a_0^{-1}+a_0a_1^{-1}+s)^{-1}=c
\end{aligned}$$

Processor	**L1(data)/L2**	**Memory**
2.4 GHz AMD Opteron	64 KB/1 MB	1 GB
1.33 GHz PowerPC G4	32 KB/512 KB	768 MB
2 GHz Intel Core Duo	32 KB/2 MB[2]	2 GB

Figure 4: List of the processors used in our evaluation.

Using this solution, we have $b_0 = (c+1)a_0^{-1}$ and $b_1 = ca_1^{-1}$. Thus, the inverse of $a = a_0 + a_1\alpha$ is $((c+1)a_0^{-1}) + (ca_1^{-1})\alpha$ and may be computed with 5 multiplications, 3 inversions and 4 additions in $GF(2^l)$. We could perform a similar boot-strapped calculation for 4-degree extensions, but chose to limit the inversion optimization to 2-degree extensions.

5 Experimental Evaluation

We have written a Galois Field library that implements $GF(2^4)$, $GF(2^8)$,$GF(2^{16})$ and $GF(2^{32})$. The core library contains code for finding irreducible polynomials, polynomial operations, and the arithmetic operations over the supported fields. Instances of Shamir's secret sharing, Reed-Solomon and algebraic signatures were also created on top of the library. The entire implementation was written in C and contains roughly 3,000 lines of code (1,000 semi-colons). This code will be made available prior to publication as a library that implements Galois fields and the aforementioned applications.

Our experimental evaluation measures the speed of multiplications as well as the throughput of three "higher-level" applications of Galois fields: Shamir secret sharing, Reed Solomon encoding and algebraic signatures. We took our measurements on three machines, whose specifications are listed in Figure 5.

Figures 5 ,6 and 7 compare multiplication throughput using the table-based techniques discussed in Section 3 over the fields $GF(2^4)$, $GF(2^8)$ and $GF(2^{16})$. These techniques are the full multiplication table (**tbl**), left-right table, (**lr_tbl**), Broder's logarithm/antilogarithm method (**lg**), the optimization chosen from [7] (**huang_lg**), and the unoptimized version of logarithm/antilogarithm (**lg_orig**).

The "uniform" bar in Figures 5, 6 and 7 represent average-case multiplication performance and measures the throughput of products with factors chosen uniformly from the Galois Field. In the "uniform" workload, the multiplicand is a monotonically increasing value masked to fit the value of a field element and the multiplier is a value chosen from a randomly generated 64K element array of field elements. The "square" bar gives the throughput from calculating squares of elements chosen with uniform probability from the Galois field, while "constant" multiplies a fixed element with a uniformly chosen random Galois field element.

In general, as expected, the "uniform" data set had lower throughput than the other workloads, for two reasons. First, the "uniform" data set draws random values from an array, which is competing with the lookup tables for a place in the cache. Second, the "uniform" workload is computing the product of two distinct elements at each step, which has a dramatic negative effect when caching large tables. The "constant" and "square" workloads only access a subset of the tables, reducing the cache competition among the lookup tables and explaining the higher throughput. These results also show that one must be attentive when measuring table-based multiplication performance. For instance, if the terms (or even a single term) of the products in a workload represent a small subset of field elements, then the reported throughput will generally higher and may not reflect the average-case performance if the real workload does not have the same characteristics as the benchmark workload.

In addition, we found that the left-right table implementation of $GF(2^8)$ appears to have the best performance for the "uniform" workload across all architectures and fields, while the rest of the results are evenly mixed across technique and architecture. As the table size grows, the performance becomes less dependent on the additional computation, explaining the cases where the throughput under the "square" and "constant" workloads are comparable to the "uniform" workload in which the extra computation becomes the bottleneck.

The effect of computation overhead is highlighted in the difference between "square" and "constant" across each method. These workloads only access a small subset of the lookup tables, thus computation overhead becomes evident. The extra computation is overshadowed by cache-related issues in the "uniform" workload; therefore, extra computation in addition to table look-up may not have a dramatic effect on average-case performance.

[2]Each core has a private 32 KB L1 cache. The L2 cache is shared between the cores.

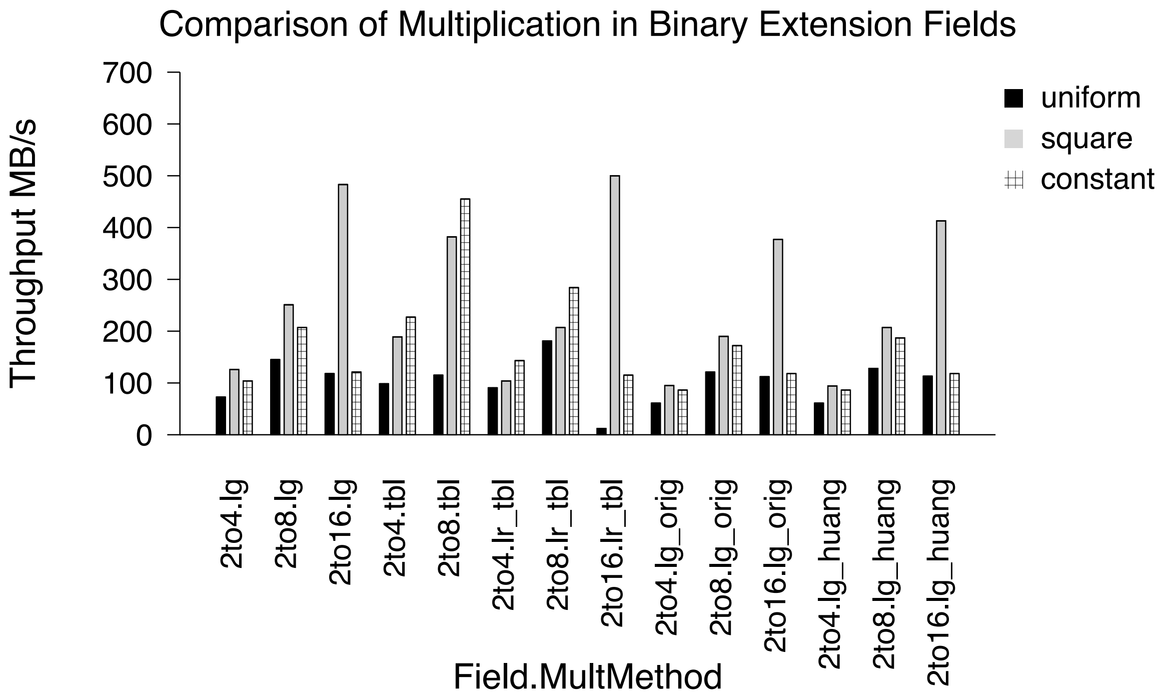

Figure 5: Throughput using multiplication tables, lg/anilg tables and LR tables over $GF(2^4)$, $GF(2^8)$ and $GF(2^{16})$on AMD.

Figure 8 compares the performance of uniform multiplications in $GF(2^{16})$ and $GF(2^{32})$. We draw particular attention to the good results of using the table method in $GF(2^8)$ when implementing $GF(2^{16})$ and $GF(2^{32})$. In fact, the $GF((2^8)^2)$ implementation either outperforms or performs comparably to $GF(2^{16})$ on all architectures, showing the effect table size and cache size can have on overall multiplication performance. The same observation is made when comparing the performance of different approaches to operations in $GF(2^{32})$: with the exception of the Intel machine, the $GF((2^8)^4)$ implementation tends to outperform $GF((2^{16})^2)$, most likely due to table size.

Overall, there is no clear winner for $GF(2^{16})$ or $GF(2^{32})$ across architectures or techniques, although, Broder's scheme and the full multiplication table technique seem to perform very well in most cases. However, it is important to note that performance degrades quickly as the size of the extension field grows from 2 to 4 degrees because of the exponential increase in the number of multiplies and the choice of irreducible polynomial.

Next, we explore the multiplication performance in a few applications: Reed-Solomon encoding, Shamir's secret sharing and algebraic signature computation. We present the results for Reed-Solomon encoding with 62 data elements and 2 parity elements in two scenarios. The first reflects basic codeword encoding, where each symbol in the codeword is an element of the appropriate Galois field. The second method, called region multiplication, performs codeword encoding over 16,384 symbols, resulting in 16,384 consecutive multiplications by the same field element. We also perform a $(3,2)$ Shamir secret split, which evaluates a random 2 degree polynomial over each field for the values 1, 2 and 3. In algebraic signatures, the signature is computed as $\{sig_{\alpha^0}(D),\ sig_{\alpha}(D)\}$ and $sig_{\beta}(D) = \sum_{i=0}^{l} d_i \beta^i$ and $|D| = l$. In addition to using the multiplication table technique shown in [16], we hand-picked α for the composite field implementations. For instance, if we choose $\alpha = 0x0101 \in GF((2^8)^2)$, then multiplication by α results in two multiplications by 1 in $GF(2^8)$, which can be optimized out as two copy operations. Note that α is chosen such that $\mathrm{ord}(\alpha) \gg b$, where b is the size of the data blocks. The signatures were calculated over 4,096 symbol blocks. Both of these optimizations could also be applied to a Reed-Solomon and Shamir implementation; for brevity, we chose to omit the optimizations.

We report the best performing multiplication method for each implementation in Figure 9. Notice that the raw multiplication performance does not always reflect the performance across the applications. For instance, as shown in

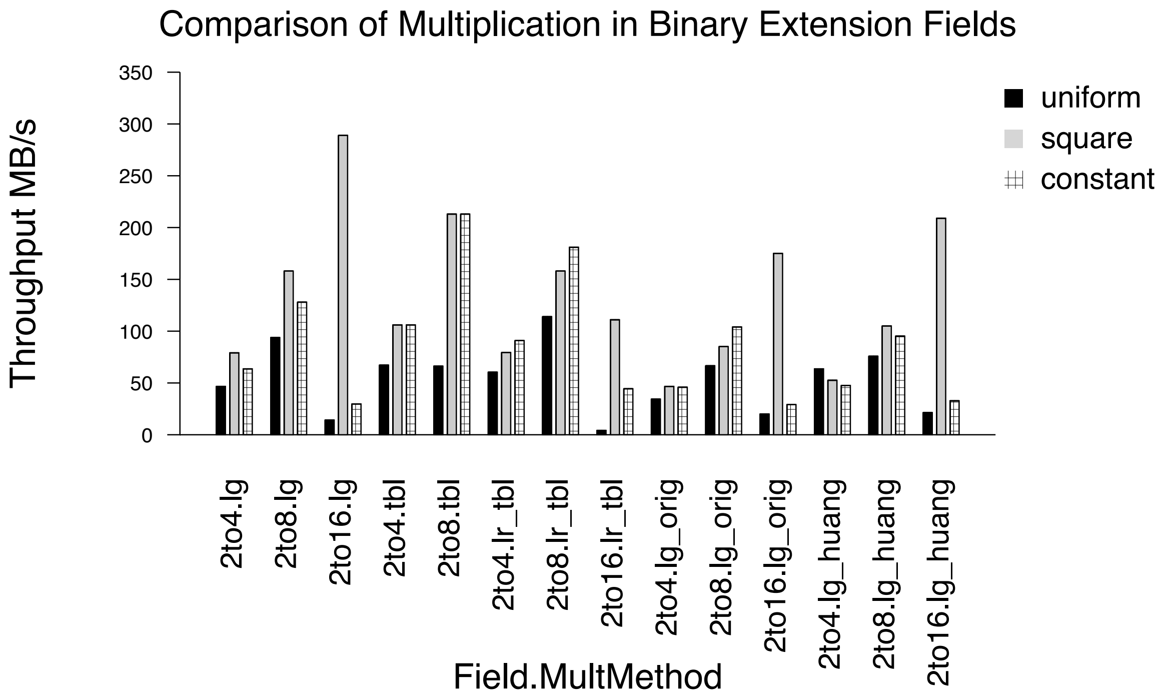

Figure 6: Throughput using multiplication tables, lg/anilg tables and LR tables over $GF(2^4)$, $GF(2^8)$ and $GF(2^{16})$ on PPC.

Figures 5 ,6 and 7, the left-right table results in the highest "uniform" multiplication throughput for $GF(2^8)$. This is not necessarily the case for fields implemented as $GF(2^8)$ in Figure 9. Cache effects are also apparent in Figure 9; due to the relatively small L1 and L2 caches in the PowerPC, the fields implemented over $GF(2^8)$ generally outperform the $GF(2^{16})$ implementations with the exception of algebraic signatures.

Overall, performance varies greatly between implementations and architecture. Interactions within the applications is much more complicated than the raw multiplication experiments and thus application must also be considered when choosing an appropriate Galois field representation. The applications are performing multiplication on sizeable data buffers, leading to more frequent cache eviction of the multiplication tables.

6 Related Work

Plank has recently released fast Galois field library [13]. The library is tailored for arithmetic in $GF(2^8)$, $GF(2^{16})$ and $GF(2^{32})$. Multiplication and division in $GF(2^8)$ and $GF(2^{16})$ are implemented using the full multiplication table and a log/antilog approach that maintains the check for zero, but optimizes the modulus operation out. Multiplication and division in $GF(2^{32})$ are implemented as either the field-element-to-bit-matrix approach [14] or a split-multiplication that uses seven full multiplication tables to byte-wise multiplication of two 32-bit words, similar to the left-right table approach described in Section 3. Our composite field approach for $GF(2^{16})$ and $GF(2^{32})$ requires less space, since we only require a single multiplication table. In addition, any ground field multiplication technique may be used in the the composite field representation.

Huang and Xu [7] presented three optimizations for the logarithm/antilogarithm approach in $GF(2^8)$. The authors show improvements to the default logarithm/antilogarithm multiplication technique that result in a 67% improvement in execution time for multiplication and division and a $3\times$ encoding improvement for Reed-Solomon. In contrast, our study evaluates a wide range of fields and techniques that may be used for multiplication and division in a variety of applications.

The emergence of elliptic curve cryptography has motivated the need for efficient field operations in extension fields [18, 1, 15]. The security of this encryption scheme is dependent on the size of the field; thus the implementations

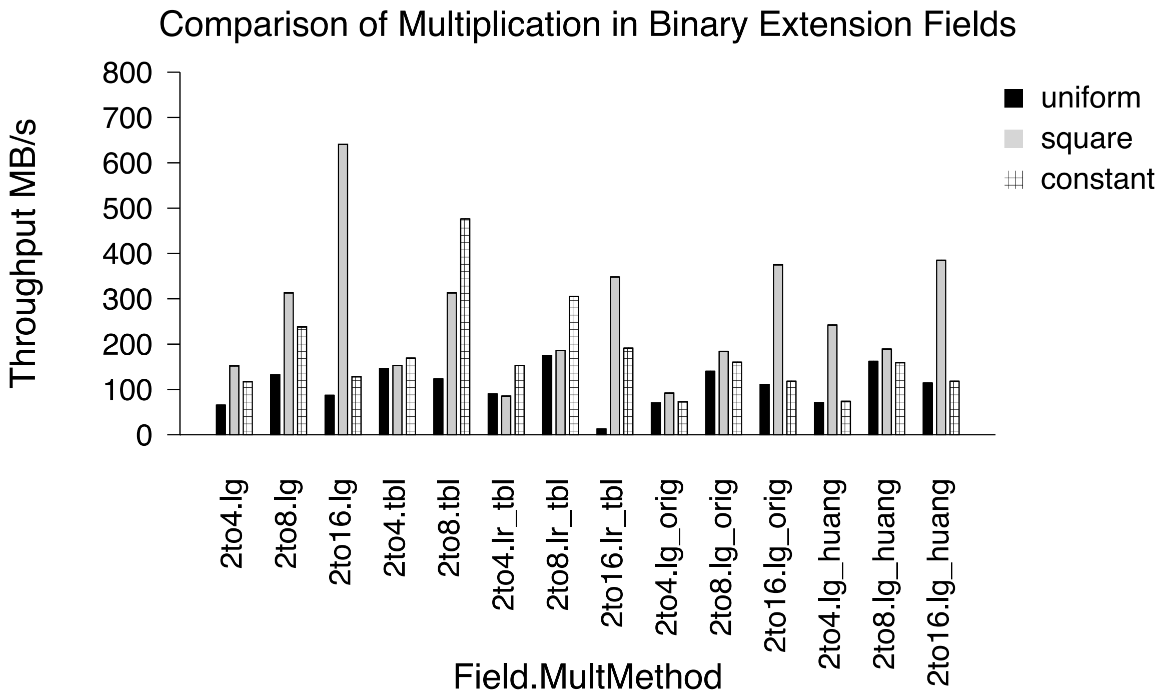

Figure 7: Throughput using multiplication tables, lg/anilg tables and LR tables over $GF(2^4)$, $GF(2^8)$ and $GF(2^{16})$ on Intel.

		$GF(2^{16})$			$GF(2^{32})$	
Architecture	Ground Field Mult.	$GF(2^{16})$	$GF((2^8)^2)$	$GF((2^4)^4)$	$GF((2^{16})^2)$	$GF((2^8)^4)$
Intel	lg	92.88	109.11	20.02	78.50	40.70
	tbl	—	103.09	23.35	—	41.54
	lr_tbl	12.69	81.56	13.20	13.16	26.97
	lg_orig	112.62	43.64	12.61	51.54	25.09
	huang_lg	112.36	74.90	13.64	75.70	28.53
AMD	lg	117.64	156.25	50.86	70.49	90.20
	tbl	—	133.33	64.89	—	112.48
	lr_tbl	12.03	150.60	38.64	10.29	76.77
	lg_orig	112.35	71.17	16.12	60.93	28.73
	huang_lg	113.38	91.91	19.51	74.65	40.26
PowerPC	lg	14.23	67.52	14.39	9.44	26.03
	tbl	—	59.26	16.82	—	30.56
	lr_tbl	4.20	57.33	13.32	2.27	24.85
	lg_orig	19.96	20.02	6.04	11.71	11.74
	huang_lg	21.35	33.07	8.86	15.13	15.82

Figure 8: Performance (in MB/s) among different implementations of $GF(2^{16})$ and $GF(2^{32})$ across different architectures.

focus on large fields (*i. e.*, of size 2^{160}). DeWin, *et al.* [18] and Savas *et al.* [15] focus on $GF((2^l)^k)$, where $gcd(l,k) = 1$. Baily and Paar [1] propose a scheme, called an Optimal Extension Field, where the field polynomial is an irreducible binomial and field has prime characteristic other than 2, leading to elements that may not be byte-aligned. In other work, Paar, *et al.* describe hardware-based composite field arithmetic [10, 11]. In most cases, research in large Galois fields is centered around cryptographic applications.

		$GF(2^8)$		$GF(2^{16})$				$GF(2^{32})$			
Architecture	Application	$GF(2^8)$		$GF(2^{16})$		$GF((2^8)^2)$		$GF((2^{16})^2)$		$GF((2^8)^4))$	
Intel	Shamir	6.40	tbl	11.29	lg	10.20	tbl	12.27	lg	10.07	tbl
	Reed-Solomon	63.85	tbl	78.84	huang_lg	48.47	tbl	41.61	lg	25.33	lr_tbl
	Region Mult.	150.00	tbl	158.72	huang_lg	119.85	tbl	130.28	lg_orig	46.03	tbl
	Algebraic Sigs	188.67	—	277.78	—	253.16	—	539.81	—	186.04	—
AMD	Shamir	14.69	tbl	22.74	huang_lg	23.25	tbl	24.30	huang_lg	21.78	tbl
	Reed-Solomon	72.18	lr_tbl	59.76	lg	87.99	tbl	58.25	huang_lg	38.39	tbl
	Region Mult.	174.00	lr_tbl	103.52	lg_orig	279.89	tbl	120.00	lg_orig	132.59	tbl
	Algebraic Sigs	283.33	—	740.74	—	348.46	—	833.33	—	645.16	—
PowerPC	Shamir	2.11	tbl	3.47	huang_lg	3.73	tbl	4.16	lg_orig	4.81	tbl
	Reed-Solomon	27.40	lg	13.68	huang_lg	21.81	lg	11.98	lg_orig	17.65	tbl
	Region Mult.	52.35	tbl	19.28	huang_lg	57.04	tbl	21.16	lg_orig	32.69	tbl
	Algebraic Sigs	158.73	—	312.94	—	232.55	—	455.06	—	337.41	—

Figure 9: Performance (in MB/s) among the best performing implementations of $GF(2^8)$, $GF(2^{16})$ and $GF(2^{32})$ for Shamir's secret sharing algorithm (encoding) , Reed-Solomon (encoding a full codeword at a time and region multiplication) and algebraic signatures (encoding). Since we are reporting the result with the highest throughput for each implementation, the multiplication table method is listed (with the exception of algebraic signatures).

7 Conclusions

We have presented and evaluated a variety of ways to perform arithmetic operations in Galois fields, comparing multiplication, inversion and application performance on three architectures. We found that the composite field representation requires less memory and, in many cases, leads to higher throughput than a binary extension field of the same size. Performance for both raw multiplications and applications shows that both CPU speed and cache size have a dramatic effect on performance, potentially leading to variation in throughput across architectures, especially for larger fields such as $GF(2^{16})$ and $GF(2^{32})$. Additionally, our results show that schemes performing well in isolation (*i. e.*, measuring multiplication throughput) may not perform as well when used in an application or to perform ground computation in a composite field. Using the approaches and evaluation techniques we have described, implementers of systems that use Galois fields for erasure code generation, secret splitting, algebraic signatures, or other techniques can increase overall system performance by selecting the best approach based on the characteristics of the hardware on which the system will run.

References

[1] D. V. Bailey and C. Paar. Optimal extension fields for fast arithmetic in public-key algorithms. *Lecture Notes in Computer Science*, 1462, 1998.

[2] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, 44(2):192–202, 1995.

[3] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An xor-based erasure-reilient coding scheme. Tech Report, 1995.

[4] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2004.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* MIT Press, Cambridge, Massachusetts, 2001.

[6] S. Gao and D. Panario. Tests and constructions of irreducible polynomials over finite fields. In Foundations of Computational Mathematics, 1997.

[7] C. Huang and L. Xu. Fast software implementation of finite field operations. Technical Report - Washington University in St. Louis, MO, 2003.

[8] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications.* Cambridge University Press, New York, NY, USA, 1986.

[9] F. J. MacWilliams and N. J. Sloane. *The Theory of Error Correcting Codes.* Elsevier Science B.V., 1983.

[10] C. Paar. A new architecture for a parallel finite field multiplier with low complexity based on composite fields. *IEEE Transactions on Computers*, 45, July 1996.

[11] C. Paar, P. Fleischmann, and P. Roelse. Efficient multiplier architectures for galois fields $gf(2^{4n})$. *IEEE Transactions on Computers*, 47, Feb 1998.

[12] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software—Practice and Experience (SPE)*, 27(9):995–1012, Sept. 1997. Correction in James S. Plank and Ying Ding, Technical Report UT-CS-03-504, U Tennessee, 2003.

[13] J. S. Plank. Fast Galois Field arithmetic library in C/C++, April 2007.

[14] J. S. Plank and L. Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *IEEE Interantional Symposium on Network Computing and Applications*, 2006.

[15] E. Savas and C. K. Koc. Efficient methods for composite field arithmetic. Tech Report, 1999.

[16] T. Schwarz, S. J. and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS '06)*, Lisboa, Portugal, July 2006. IEEE.

[17] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, Nov. 1979.

[18] D. Win, Bosselaers, Vandenberghe, D. Gersem, and Vandewalle. A fast software implementation for arithmetic operations in $GF(2^n)$. In *ASIACRYPT: International Conference on the Theory and Application of Cryptology*, 1996.

[19] L. Xu and J. Bruck. X-code : MDS array codes with optimal encoding. In *IEEE Transactions on Information Theory*, 1999.