Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications

Kevin M. Greenan
Univ. of California, Santa Cruz
kmgreen@cs.ucsc.edu

Ethan L. Miller
Univ. of California, Santa Cruz
elm@cs.ucsc.edu

Thomas J. E. Schwarz, S.J.
Santa Clara University
tjschwarz@scu.edu

Abstract

Galois field implementations are central to the design of many reliable and secure systems, with many systems implementing them in software. The two most common Galois field operations are addition and multiplication; typically, multiplication is far more expensive than addition. In software, multiplication is generally done with a look-up to a pre-computed table, limiting the size of the field and resulting in uneven performance across architectures and applications.

In this paper, we first analyze existing table-based implementation and optimization techniques for multiplication in fields of the form $GF(2^l)$. Next, we propose the use of techniques in composite fields*: extensions of $GF(2^l)$ in which multiplications are performed in $GF(2^l)$ and efficiently combined. The composite field technique trades computation for storage space, which prevents eviction of look-up tables from the CPU cache and allows for arbitrarily large fields. Most Galois field optimizations are specific to a particular implementation; our technique is general and may be applied in any scenario requiring Galois fields. A detailed performance study across five architectures shows that the relative performance of each approach varies with architecture, and that CPU, memory limitations and fields size must be considered when selecting an appropriate Galois field implementation. We also find that the use of our composite field implementation is often faster and less memory intensive than traditional 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 erasure encoded symbols, evaluate and interpolate polynomials in Shamir's secret sharing algorithm [22], compute algebraic signatures over variable-length strings of symbols [21], and encrypt blocks of data in the current NIST advanced encryption standard [14]. Current memory, CPU cache sizes and preferred approaches limit most applications to performing computation in either $GF(2^8)$ or $GF(2^{16})$. The goal of our research is to study the multiplication performance of these common fields, propose an alternate representation for arbitrary-sized fields and compare performance across all representations on different CPU architectures and for different workloads.

Multiplication in $GF(2^l)$ is usually done using precomputed 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, multiplication has the greatest effect on overall algorithm performance because a table look-up is more expensive than bit-wise XOR. Due to the restrictions of look-up tables, elements in the implemented field are almost always smaller than a computer word, while bit-wise XOR operates over words by definition. In many applications, the multiplication operation is used just as often as addition; 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. This field only has 256 elements, which results in small multiplication tables; however, use of $GF(2^8)$, for example, restricts the size of a Reed-Solomon codeword to no more than 257 elements [13]. The smallest feasible field larger than $GF(2^8)$ is $GF(2^{16})$. Growing to $GF(2^{16})$ and beyond has a significant impact on multiplication and other field operations. 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 requiring 256 KB of memory, which may fit in the standard L2 cache but will likely be only partially resident in an L1 cache. Using a straightforward table-based multiplication approaches to match word size in a 32-bit system (e.g. $GF(2^{32})$) is impossible, given modern memory sizes.

This paper appeared in the *Proceedings of the 16th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2008)*, Baltimore, MD, September 2008.

Composite fields are an alternative to traditional 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*. This technique trades additional computation for a significant decrease in storage space relative to traditional table-based methods. Since the cost of a cache miss is comparable to that of the execution of many instructions, trading additional computation for lower storage requirements can significantly increase performance on modern processors. Additionally, many applications will use Galois fields for different purposes; using the composite field technique, it may be possible to reuse the the ground Galois field tables for several variable-sized Galois field extensions.

The performance of the Galois field implementation is a critical factor in overall system performance. The results presented here show dramatic differences in throughput, but there is 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 architecture, CPU speed and cache size. In this paper, we restrict our investigation to the 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 for future work.

The contributions of this paper are threefold. First, we present and compare popular table-based binary extension field implementation techniques and some optimizations. Next, we propose the use of software-based composite fields when implementing $GF(2^{16})$ and $GF(2^{32})$. Finally, we show that the performance of different implementations of Galois fields is highly dependent on underlying hardware and workload, and suggest techniques better optimized for distinct architectures. Unlike many hardware and software Galois field optimizations, all of the techniques described are general and may be applied to any application requiring Galois field arithmetic.

2. Applications of Galois Fields

The use 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. Many 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 community [5, 3, 24]. 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 [17], are MDS. As a result, Reed-Solomon codes provide flexibility and optimal storage efficiency. A Reed-Solomon codeword is computed by generating m parity symbols from k data symbols such that any m erased symbols may be recovered. Each parity symbol is generated using k Galois field multiplications and $k-1$ additions. The required size of the underlying Galois field is bound by the number of parity and data symbols in a codeword. Linear erasure codes are also used in network coding to maximize information flow in a network, where linear combinations of symbols are computed (mixed) at intermediate nodes within a network [1]. The field size is typically bound by network size and connectivity.

Threshold cryptography algorithms, such as Shamir's secret sharing algorithm [22], also rely on Galois fields for encoding and decoding. The algorithm chooses a random k-degree polynomial over a Galois field; the zero-th coefficient is the 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 zero-th 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 [21]. Algebraic signatures are *Rabinesque* because of the similarity between signature calculation and the hash function used in the Rabin-Karp string matching algorithm [6]. 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.

While the use of fields larger than $GF(2^8)$ is generally considered overkill, there are practical instances where large fields are required. As long as there are at most 257 symbols in a codeword, Reed-Solomon can be implemented with $GF(2^8)$. Once the number of symbols exceeds this bound, a larger field must be used. For example, the disaster recovery codes described in [9] may require thousands of symbols per codeword; thus, a larger field such as $GF(2^{16})$ must be used. Algebraic signatures have a similar restriction. In this case case, the length of the string is lim-

ited by the size of the underlying field and in many cases a field larger than $GF(2^8)$ is required.

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. However, as storage systems begin to use non-volatile memories [8], Galois field performance may begin to dominate storage and retrieval time. 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. As with any field, addition and multiplication are associative, distributive and commutative [12]. 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; however, multiplication is more complicated. In order to multiply two elements $a, b \in GF(2^l)$, we perform polynomial multiplication of $a(x) \cdot b(x)$ and reduce the product modulo an 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 α, $\text{ord}(\alpha)$, 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.

This section describes several approaches to performing multiplication over the fields $GF(2^4)$, $GF(2^8)$, and $GF(2^{16})$ and presents several optimizations. All of the methods described here may be used to perform ground field calculations in the composite field representation.

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

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 precomputed 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—$O(n^2)$ for fields of size n.

Input : $a, b \in GF(2^l)$ and $FLD\_SIZE = 2^l$
if a is 0 OR 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

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 requires $O(n)$ space for fields of size n and 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 $\beta = \text{antilog}(i)$. As shown in Figure 1, the product of two non-zero elements $a, b \in GF(2^l)$ can be computed as $a \cdot b = \text{antilog}(\log(a) + \log(b)) \mod 2^l - 1$.

3.2. Optimization of the Full Multiplication Table

The size of the full multiplication table is reduced by breaking a multiplier in $GF(2^l)$ into a left and a right part. The result of multiplication by a left and by a right part is stored 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" [21]; 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 $a(x) = a_1 + a_2 x + \cdots + a_{l-1}x^{l-1}$ and $b(x) = b_1 + b_2 x + \cdots + b_{l-1}x^{l-1}$ 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 $2^{l/2} \cdot 2^l$ entries each, assuming l is even. This approach, which computes the product of two elements using two lookups, a bit-

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

wise 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 $mult\_tbl\_left[a_1 >> l/2][b]$ 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.

This multiplication table optimization is highly effective for $GF(2^8)$ and $GF(2^{16})$. The standard multiplication table for $GF(2^8)$ requires 64 KB, which typically fits 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 is much more likely to be fully resident 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 Log/Antilog Method

While our previous optimization traded an additional calculation for space savings, the optimizations for the log/antilog method 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 $2^l - 1$ as well. 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 quintupled in size, but the product of a and b may now be calculated simply as $a \cdot b = \text{antilog}[\log[a] + \log[b]]$. We call this approach the optimized logarithm/antilogarithm or Broder's scheme.

Technique	Space	Complexity
Mult. Table	$2^l \cdot 2^l$	1 LOOK
Log/Antilog	$2^l + 2^l$	3 LOOK, 2 BR, 1 MOD 1 ADD
Log/Antilog Optimized	$5 \cdot 2^l$	3 LOOK, 1 ADD
Huang and Xu	$2^l + 2^l$	3 LOOK, 1 BR, 3 ADD 1 SHIFT, 1 AND
LR Mult. Table	$2^{(3l/2)+1}$	2 LOOK, 2 AND 1 XOR, 1 SHIFT

Table 1: Ground field memory requirements and computation complexity of multiplication in $GF(2^l)$. The operations are abbreviated LOOK for table lookup, BR for branch and MOD for modulus; the rest refer to addition and the corresponding bitwise operations.

Huang and Xu proposed three improvements to the log/antilog approach [11]. 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

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

Due to the similarity between the second and third improvements in Huang and Xu [11] and Broder's scheme, we chose to only include the first improvement (called Huang and Xu) for comparison in our study.

This section has described a variety of techniques and optimizations for multiplication in $GF(2^l)$; Table 1 lists the space and computation requirements for each approach. 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 specialized composite field techniques [15, 16], in which multiplication in a large Galois field is implemented in terms of a smaller Galois field. While only using fields with sizes that are a power of 2, we want to implement multiplication and division in $GF(2^n)$ in terms of $GF(2^l)$, where $n = l \cdot k$. Galois field theory states that $GF(2^n)$ is isomorphic to an extension field of $GF(2^l)$ generated by an irreducible polynomial $f(x)$, 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 $a_{k-1} \cdot x^{k-1} + \ldots + a_2 \cdot x^2 + a_1 \cdot x + a_0$ 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 $f(x)$. Their product is

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

For $i > k-1$ in this expression, we replace x^i with $x^i \bmod f(x)$, perform the multiplication, and reorganize by 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 (*i. e.*, α such that $f(\alpha) = 0$) and irreducibility testing is simple. Otherwise, the Ben-Or algorithm [7] is an efficient way to find irreducible polynomials.

We have implemented multiplication in $GF((2^l)^2)$, for $l \in \{4, 8, 16\}$ and $GF((2^l)^4)$, for $l \in \{4, 8\}$ using the composite field representation, as described in the remainder of this section We have developed a similar approach for computing the inverse of an element. The composite field inversion techniques and their performance are discussed in the full version of this paper [10].

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 are well-suited 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

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

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. 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`, resulting in an irreducible polynomial that is 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 in this manner.

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 $GF((2^8)^2)$; fortunately, 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 $GF(2^{32})$ using $5 \cdot 5 = 25$ 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 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 $x^4 + x^2 + sx + t$ ($s, t \notin \{0, 1\}$), 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 $a_3 b_3$, $a_3 b_2 + a_2 b_3$, and by $a_3 b_1 + a_2 b_2 + a_1 b_3$, reusing some of the results. For instance, we have an addend of $a_3 b_3 (t+1) x^2$ and of $a_3 b_3 t$, 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 $GF(2^{16})$.

5. Experimental Evaluation

We have written a Galois field library that implements $GF(2^4)$, $GF(2^8)$,$GF(2^{16})$ and $GF(2^{32})$ using the approaches presented in Sections 3 and 4. 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

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[1]	2 GB
2 GHz Intel Pentium 4 M	8 KB/512 KB	1 GB
400 MHz ARM9	32 KB/—	128 MB

Table 2: List of the processors used in our evaluation.

of the library. The entire implementation was written in C and contains roughly 3,000 lines of code. 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 five machines, whose specifications are listed in Table 2. The remainder of this section focuses on 8 key observations found when studying the effect of Galois field implementation on performance. Due to space, we only present a subset of the results in this section. For more detailed performance numbers, please refer to [10].

5.1. Performance using $GF(2^l)$

Figures 3(a)–3(d) 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**), optimized logarithm/ antilogarithm method (**lg**), the optimization chosen from [11] (**huang_lg**), and the unoptimized version of logarithm/antilogarithm (**lg_orig**).

Three distinct workloads were run on four of the architectures. Although they are artificial, the workloads embody typical operations in Galois fields. The UNIFORM workload represents the average-case by computing the product of a randomly-chosen field element (drawn from an array) and a monotonically-increasing value masked to fit the value of a field element. The CONSTANT workload computes the product of a constant value and a randomly chosen field element. The SQUARE workload squares a randomly chosen element, then squares the result and so forth. We expected the UNIFORM workload to essentially use the entire look-up table, while CONSTANT and SQUARE typically utilize only a subset of a table.

Observation #1 : *Workload determines look-up table access pattern, which affects performance.*

As expected, the UNIFORM data set has lower throughput than the other workloads, for two reasons. First, UNIFORM draws random values from an array, competing with the look-up tables for space in the cache. Second, the uniform workload computes the product of two distinct elements at each step, which has a dramatic effect when caching large tables. Unlike UNIFORM , the CONSTANT and SQUARE workloads only access a subset of the tables, which results in reduced cache competition and higher throughput. Our results show that one must be attentive when measuring table-based multiplication performance, since look-up table access patterns have a dramatic effect on performance.

Observation #2 : *Cache size affects performance.*

In addition to cache competition, the actual size of the look-up table relative to cache size also greatly affects performance. For example, the full multiplication table for $GF(2^8)$ generally performs worse than any other algorithm for the average-case workload, since a full multiplication table requires at least 64 KB, and thus does not fit in the L1 cache of most processors.

Observation #3 : *Performance decreases as the cache-resident portion of the table decreases.*

SQUARE and CONSTANT perform much better than UNIFORM when the table size is quite large, since a smaller fraction of the multiplication table is required in cache. When the table size is relatively small, the performance of SQUARE and CONSTANT is closer to UNIFORM , since a larger fraction of the multiplication table will be kept in cache. In addition, any extra computation is overshadowed by cache-related issues in the UNIFORM workload; extra computation in addition to table look-ups has little effect on average-case performance.

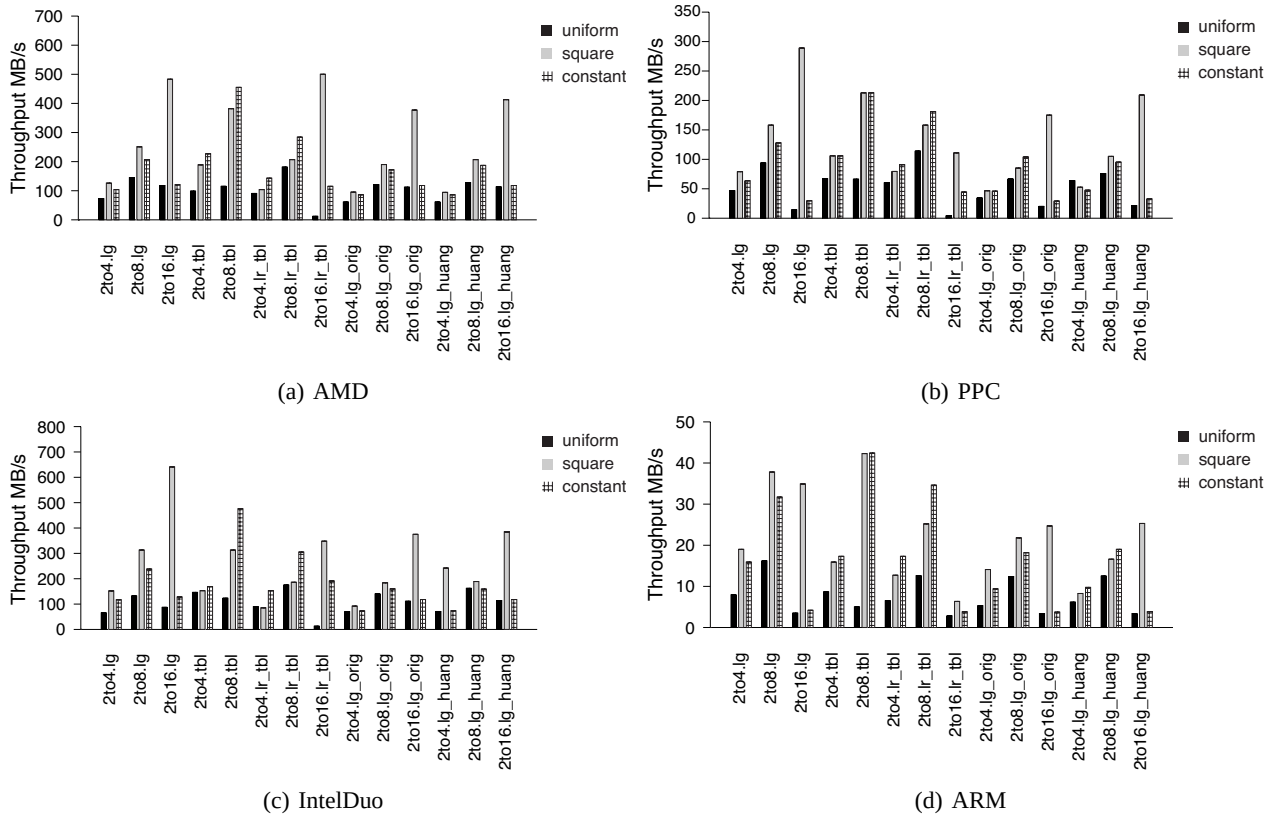
Overall, we found that the left-right implementation of $GF(2^8)$ appears to have the best performance for the UNIFORM workload across three of the four architectures. This optimization appears to provide the best combination of table size and computation for these architectures. We believe that the optimized logarithm/antilogarithm approach for $GF(2^8)$ performs best on ARM due to the computational constraints on that processor and the lack of an L2 cache.

5.2. Performance using $GF((2^l)^k)$

Figure 5 shows the normalized multiplication throughput (in MB/s) of the composite field technique and the traditional look-up table techniques for $GF(2^{16})$ and $GF(2^{32})$ over the UNIFORM workload. In general, the logarithm/antilogarithm approaches performed the best across both composite field and traditional look-up table techniques, thus we omit the other approaches.

Figures 4(b)–4(d) show the multiplication performance in three typical applications: Reed-Solomon encoding, Shamir 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, in which each symbol in the codeword is an element of the appropriate Galois

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



(a) AMD

(b) PPC

(c) IntelDuo

(d) ARM

Figure 3: Throughput of ground fields using multiplication tables, lg/anilg tables and LR tables. The labels on the x-axis are given as *field.multmethod*, where *field* is the underlying field and *multmethod* is the multiplication algorithm.

field. The second method, called *region multiplication*, performs codeword encoding over 16K symbols, resulting in 16K 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 by $\{sig_{\alpha^0}(D),\ sig_{\alpha}(D)\}$ and $sig_{\beta}(D) = \sum_{i=0}^{l} d_i \beta^i$ and $|D| = l$ over 4K symbol blocks. Both Shamir and algebraic signatures use Horner's scheme for polynomial evaluation [6], so most of the multiplications are reminiscent of CONSTANT .

We explored an interesting optimization specific to the composite field representation when computing algebraic signatures. In addition to using the multiplication table technique shown in [21], we hand-picked α for the composite field implementations. For instance, if we choose $\alpha = \texttt{0x0101} \in GF((2^8)^2)$, 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 $\text{ord}(\alpha) \gg b$, where b is the size of the data blocks. This optimization could also be applied to a Reed-Solomon and Shamir implementation; for brevity, we omit the optimizations.

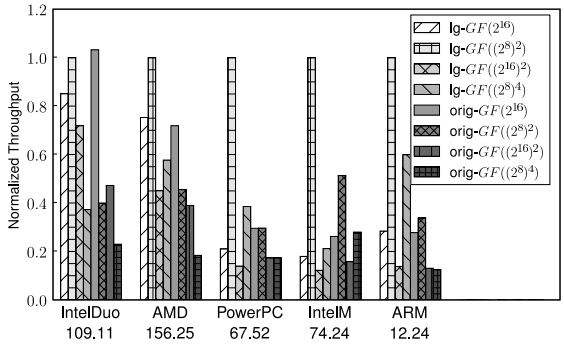
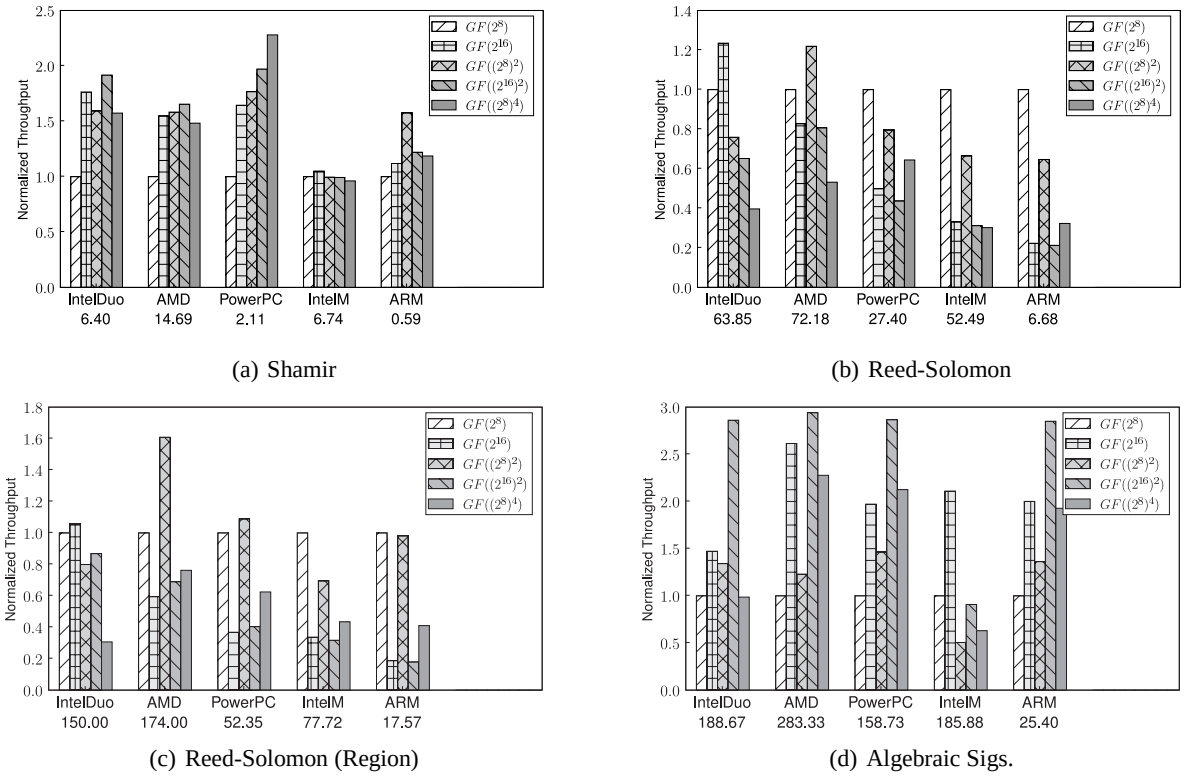


Figure 5: Normalized throughput of the traditional look-up algorithms and the composite field method. All values are normalized to lg-$GF((2^8)^2)$. The actual lg-$GF((2^8)^2)$ throughput numbers (MB/s) are shown on the x-axis along with architecture.

We ran all of the techniques (traditional look-up and composite field representations) for each application. We report the "best" performer for each application and field.

Observation #4 : *The composite field representation is quite effective and in some cases outperforms the traditional look-up table techniques.*



(a) Shamir

(b) Reed-Solomon

(c) Reed-Solomon (Region)

(d) Algebraic Sigs.

Figure 4: Normalized throughput of various applications. All values are normalized to $GF(2^8)$. The actual throughput numbers (MB/s) are shown on the x-axis along with architecture. All of the reported numbers are given for the *best* performing technique for the particular application.

Figure 5 compares the normalized throughput of UNIFORM multiplications in $GF(2^{16})$ and $GF(2^{32})$, where the ground field operations are computed using the unoptimized and optimized logarithm/antilogarithm approaches. We observed very good results when using $GF(2^8)$ as a ground field in the $GF(2^{16})$ and $GF(2^{32})$ composite field implementations. 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 effect of cache size is very apparent when comparing IntelDuo and AMD to the other, relatively cache-constrained, architectures. We notice a striking performance improvement when implementing $GF(2^{16})$ as $GF((2^8)^2)$ in the architectures with smaller L1 and L2 caches. This improvement is largest on the IntelM processor, which only has an 8 KB L1 cache.

Observation #5 : *Raw multiplication performance does not always reflect application performance.*

We report the best performing multiplication method for each application in Figure 4. While not shown in Figure 4, the raw multiplication performance does not always reflect the performance across the applications. For instance, as shown in Figure 3, the left-right table results in the highest UNIFORM multiplication throughput for $GF(2^8)$. This is not necessarily the case for applications implemented using $GF(2^8)$ in Figure 4.

Overall, performance varies greatly between implementations and architecture. Interactions within the applications is much more complicated than the raw multiplication experiments; thus, the application must also be considered when choosing an appropriate Galois field representation. For example, the applications in Figure 4 perform multiplication on sizeable data buffers, leading to more frequent cache eviction of the multiplication tables.

Observation #6 : *Composite field implementation is affected by cache size.*

Cache effects are apparent in Figure 4. Due to the relatively small L1 and L2 caches in the PowerPC, IntelM and ARM, the fields implemented over $GF(2^8)$ generally outperform the $GF(2^{16})$ implementations in both Reed-Solomon encoding algorithms. The multiplication workload of algebraic signatures and Shamir's secret sharing algorithm is similar to CONSTANT , thus computation becomes a limiting factor.

Observation #7 : *No technique is best for every architecture and application.*

There exists no clear winner for $GF(2^{16})$ or $GF(2^{32})$ across architectures or techniques. 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 multiplications and the choice of irreducible polynomial. While there is variance in the performance across architecture and application, the optimized logarithm/antilogarithm scheme and the full multiplication table technique for $GF(2^8)$ seem to perform very well in most cases.

Observation #8 : *Application-specific optimizations for composite fields can further improve performance.*

As shown in Figure 4(d), hand-picking field elements when computing algebraic signatures in a composite field can further improve performance. As an example, we find that the $GF((2^{16})^2)$ implementation with $\alpha = \texttt{0x00010001}$ outperforms all others. Given a large ground field (*i. e.*, $GF(2^{16})$ or larger), where the elements are expected to have higher order, we should be able to perform a similar optimization for a Reed-Solomon implementation.

5.3. Discussion

While we cannot state a sweeping conclusion based on our results, there are a few interesting points worth mentioning when considering composite fields. First, one advantage to using composite fields is the ability to optimize for an application based on the underlying field representation. We have given an example of how this is done for algebraic signatures, noting that similar optimizations are possible for Reed-Solomon and Shamir's secret sharing algorithm. Second, the composite field representation has utility even when the application only requires $GF(2^8)$. As we have shown in our analysis, there exist cases where $GF((2^8)^2)$ outperforms $GF(2^8)$. This is quite evident when using the AMD processor, since it has the fastest clock speed and largest L1 cache. Finally, if an implementation requires a field larger than $GF(2^8)$, the composite field representation may lead to better performance. In many cases, $GF((2^8)^2)$ tends to outperform $GF(2^{16})$ and $GF((2^{16})^2)$ is expected to outperform many software-based $GF(2^{32})$ implementations.

6. Related Work

Plank has recently released a fast Galois field library [18], tailored for arithmetic in $GF(2^8)$, $GF(2^{16})$ and $GF(2^{32})$. Multiplication in $GF(2^8)$ and $GF(2^{16})$ is implemented using the full multiplication table and a log/antilog approach that maintains the check for zero, but optimizes the modulus operation out. Multiplication in $GF(2^{32})$ is implemented as either the field-element-to-bit-matrix approach [19] or a split-multiplication that uses seven full multiplication tables for multiplication of two 32-bit words. 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.

A great deal of effort has gone into alternative Galois field representations for Reed-Solomon codes. One popular optimization uses the bit-matrix representation, and is used in Reed-Solomon erasure coding [19, 4], where each multiplication over the field $GF(2^l)$ is transformed into a product of an $l \times l$ bit matrix and an $l \times 1$ bit vector. This scheme has the advantage of computing all codewords using nothing more than word-sized XORs, by trading a multiplication for at least l XOR operations. The bit-matrix representation is typically stored as a look-up. For Reed-Solomon, $k \times m$ matrices must be stored for a code that computes m parity elements from k data elements, where $k \times m$ is generally less than 256. This optimization works well for many Reed-Solomon implementations, but it may be difficult or impossible to map the optimization to other applications. Additionally, the bit-matrix scheme may not work well in cases where all field elements are represented or there is relatively little data to encode.

Huang and Xu presented three optimizations for the logarithm/antilogarithm approach in $GF(2^8)$ [11]. The authors show improvements to the default logarithm/antilogarithm multiplication technique that result in a 67% improvement in execution time for multiplication 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 in a variety of applications.

The emergence of elliptic curve cryptography has motivated the need for efficient field operations in extension fields [23, 2, 20]. The security of this encryption scheme is dependent on the size of the field; thus the implementations focus on large fields (*i. e.*, of size 2^{160}). DeWin, *et al.* [23] and Savas, *et al.* [20] focus on $GF((2^l)^k)$, where $gcd(l,k) = 1$. Baily and Paar [2] 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 [15, 16].

7. Conclusions

We have presented and evaluated a variety of ways to perform arithmetic operations in Galois fields, comparing multiplication, and application performance on five distinct architectures and six workloads: three artificial workloads and three actual Galois field-based coding applications. 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 high 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.

The choice of Galois field calculation method for a particular application resulted in differences as high as a factor of three for different approaches on the same architecture; moreover, no single approach worked best for any given architecture across applications. Using the approaches and evaluation techniques we have described, implementers of systems that use Galois fields for erasure code generation, secret sharing, 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 and the application-generated Galois field arithmetic workload.

References

[1] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, Jul 2000.

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

[3] 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.

[4] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical report, ICSI, UC-Berkeley, 1995.

[5] 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.

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

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

[8] K. M. Greenan and E. L. Miller. PRIMS: Making NVRAM suitable for extremely reliable storage. In *Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep '07)*, June 2007.

[9] K. M. Greenan, E. L. Miller, T. J. E. Schwarz, and D. D. Long. Disaster recovery codes: increasing reliability with large-stripe erasure correcting codes. In *StorageSS '07*, pages 31–36, New York, NY, USA, 2007. ACM.

[10] K. M. Greenan, E. L. Miller, and T. J. E. Schwarz, S.J. Analysis and construction of Galois fields for efficient storage reliability. Technical report, UC-Santa Cruz, 2007.

[11] C. Huang and L. Xu. Fast software implementation of finite field operations. Technical report, Washington Univ., 2003.

[12] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge Univ. Press, New York, USA, 1986.

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

[14] National Institute of Standards and Technology. *FIPS Publication 197 : Advanced Encryption Standard*.

[15] 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.

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

[17] 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.

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

[19] 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.

[20] E. Savas and C. K. Koc. Efficient methods for composite field arithmetic. Technical report, Oregon St. Univ., 1999.

[21] 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.

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

[23] 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.

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