# Efficient Software Implementations of Large Finite Fields $GF(2^n)$ for Secure Storage Applications

JIANQIANG LUO
Wayne State University
KEVIN D. BOWERS, and ALINA OPREA
RSA Laboratories
LIHAO XU
Wayne State University

Finite fields are widely used in constructing error-correcting codes and cryptographic algorithms. In practice, error-correcting codes use small finite fields to achieve high-throughput encoding and decoding. Conversely, cryptographic systems employ considerably larger finite fields to achieve high levels of security. We focus on developing efficient software implementations of arithmetic operations in reasonably large finite fields as needed by secure storage applications.

In this paper, we study several arithmetic operation implementations for finite fields ranging from $GF(2^{32})$ to $GF(2^{128})$. We implement multiplication and division in these finite fields by making use of precomputed tables in smaller fields, and several techniques of extending smaller field arithmetic into larger field operations. We show that by exploiting known techniques, as well as new optimizations, we are able to efficiently support operations over finite fields of interest. We perform a detailed evaluation of several techniques, and show that we achieve very practical performance for both multiplication and division.

Finally, we show how these techniques find applications in the implementation of HAIL, a highly available distributed cloud storage layer. Using the newly implemented arithmetic operations in $GF(2^{64})$, HAIL improves its performance by a factor of two, while simultaneously providing a higher level of security.

Categories and Subject Descriptors: E.4 [**Coding and Information Theory**]: Error control codes; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effectiveness)*; I.1.2 [**Computing Methodologies**]: Algorithms—*Algebraic algorithms*

General Terms: Algorithms, Performance, Security

Additional Key Words and Phrases: Finite field arithmetic, cloud storage systems, cryptographic algorithms

Author's address: J. Luo, Wayne State University, 5057 Woodward Avenue, Detroit, MI, 48202; K. D. Bowers, A. Oprea, RSA Laboratories, 11 Cambridge Center, Cambridge, MA, 02142; L. Xu, Wayne State University, 5057 Woodward Avenue, Detroit, MI, 48202.

## 1. INTRODUCTION

Finite fields are widely used in constructing error-correcting codes and cryptographic algorithms. For example, Reed-Solomon codes [Reed and Solomon 1960] are based on arithmetic operations in finite fields. Various cryptographic constructions, including the Diffie-Hellman key exchange protocol [Diffie and Hellman 1976], discrete-log based cryptosystems (e.g., El-Gamal encryption [ElGamal 1985], DSA signatures [Kravitz 1993]), and schemes based on elliptic curves [Miller 1986] are implemented in finite fields of large prime order. While practical implementations of error-correcting or erasure codes use small finite fields to achieve high-throughput encoding and decoding, cryptographic systems need considerably larger finite fields for high security guarantees.

In this paper, we provide efficient implementations of arithmetic operations for finite fields of characteristic two, ranging from $GF(2^{32})$ to $GF(2^{128})$. The main reason is that finite fields within this range are very suitable for secure data storage applications and systems. Most storage systems today employ erasure coding based on small finite fields (e.g., $GF(2^8)$ or $GF(2^{16})$) to provide fault tolerance in case of benign failures (for instance, drive crashes). They achieve efficiency through the use of small finite fields, but they have not been designed to sustain adversarial failures. With the advent of cloud storage, offered by providers such as Amazon S3 and others, a whole host of new failure models need to be considered, e.g., mis-configuration, insider threats, software bugs, and even natural calamities. Accordingly, storage systems have to be redesigned with robustness against adversarial failures.

One direct consequence is that reasonably larger finite fields are needed to realize both fault tolerance and security of storage systems. In general, the larger a finite field is, more security it offers. Compared to general cryptographic operations, though, for data storage applications, a finite field of size $GF(2^{64})$ or $GF(2^{128})$ is considered to be large enough to achieve desired security degree, while not imposing too much computational cost for other operations, such as erasure coding for data reliability. We will show such an example, the HAIL system, later in this paper. Thus throughout this paper, our focus will be on finite fields up to $GF(2^{128})$.

To efficiently implement operations over finite fields of the form we are interested in, we combine well established techniques with novel optimizations. Currently, several methods are most commonly used for implementing finite field arithmetic. The *binary polynomial method* represents finite field elements as polynomials and translates field arithmetic operations into corresponding operations on polynomials. While addition and subtraction are extremely fast (as they can be implemented with exclusive-or operations), polynomial multiplication and division involve operations known to be inefficient in fields of large order (e.g., modular reduction modulo an irreducible polynomial or finding polynomial inverses). With additional optimizations (precomputation of lookup tables), the binary polynomial method is nevertheless very efficient in small fields. For operations on larger finite fields, the *extension field method* [Win et al. 1996] uses precomputed tables in the base field and several techniques [Huang and Xu 2003; Win et al. 1996] for extending small field arithmetic into larger field operations.

In our algorithms, we use the extension field arithmetic method together with novel optimizations for operations in the base field (of small size). We propose to use certain irreducible polynomial patterns that reduce the complexity of modular reduction, and then we specifically design an efficient division algorithm for those polynomial patterns in the base field. We also use precomputed log and antilog tables and a new method of caching table lookup results for optimizing the multiplication operation in the base field. Using these techniques, we provide several optimized implementations of multiplication and division operations in such fields, compare their performance on various platforms with that of best known arithmetic operations, and show the practical benefits of our optimizations.

Finally, we show how our work impacts the performance of HAIL (High Availability and Integrity Layer) [Bowers et al. 2009], a distributed cryptographic cloud storage system, which directly motivated this work. HAIL provides a new way of building reliable cloud storage out of unreliable components, by extending the RAID principle [Patterson et al. 1988] into the cloud. HAIL disperses files across cloud providers using Reed-Solomon codes applied to file stripes. Since providers are untrusted entities, the integrity of file blocks is protected by message-authentication codes (MACs).

To reduce the amount of additional storage needed for integrity checks, HAIL introduces a new cryptographic construction that embeds the MACs within the parity blocks of each stripe. For our implementation of HAIL, we aim to achieve 64-bit security. We can accomplish this by implementing Reed-Solomon codes in $GF(2^{64})$ using our optimized techniques for arithmetic operations in large Galois fields. It is possible to obtain the same level of security with arithmetic operations in smaller fields, at the cost of slower encoding and decoding performance. We discuss in Section 6 that the cost of achieving 64-bit security using operations in $GF(2^{32})$ is equivalent to performing four encodings in $GF(2^{32})$. We then show that by utilizing the newly implemented operations in $GF(2^{64})$, we can achieve 64-bit security while improving the encoding and decoding performance of HAIL by a factor of two compared to an implementation based on 32-bit operations.

To summarize, the contributions of our paper include:

(1) We survey the major efficient algorithms that could be used for implementing arithmetic operations over reasonably large finite fields, as needed by secure storage applications.
(2) We provide several implementations of arithmetic operations in large finite fields of characteristic two by extending existing methods with newly proposed optimizations.
(3) We extensively evaluate and compare different implementations on multiple platforms and show which ones perform best under specific conditions.
(4) We show how our implementation of 64-bit arithmetic greatly impacts the encoding and decoding performance of the HAIL distributed cloud storage protocol.

## 2. RELATED WORK

A lot of cryptographic algorithms are based on finite field arithmetic [Diffie and Hellman 1976; ElGamal 1985; Miller 1986; Kravitz 1993]. Finite fields used in the design of cryptographic primitives can be classified into three types: prime fields, binary fields, and optimal extension fields. Prime fields can be represented by $\mathbb{F}_p$, where $p$ is a prime number. Binary fields are fields of characteristic two $\mathbb{F}_{2^n}$ or $GF(2^n)$, where $n$ is an integer number greater than 0. Optimal extension fields are fields of the form $\mathbb{F}_{p^n}$, where $p$ is a prime number and $n$ is an integer number that has to satisfy some restrictions with respect to $p$ [Bailey and Paar 1998]. Due to differences in the algebraic structure of these finite fields, the arithmetic operations in different types of fields are also implemented differently. Guajardo et al. presented a survey of efficient software implementations for general field arithmetic [Guajardo et al. 2006]. In this paper, we focus on binary fields.

There are efficient multiplication and division approaches for general binary fields. Lopez et al. [López and Dahab 2000] introduced several multiplication algorithms for $GF(2^n)$. The algorithms include the *right-to-left comb method*, the *left-to-right comb method*, and the *left-to-right comb method with windows of width $w$*. These algorithms have been shown to greatly outperform the traditional shift-and-add method [Hankerson et al. 2000], and they are among the fastest existing multiplication algorithms. Widely used efficient algorithms for division include the *Extended Euclidean Algorithm* and its two variants: the *Binary Extended Euclidean Algorithm* [Menezes et al. 1997] and the *Almost Inverse Algorithm* [Schroeppel et al. 1995]. These algorithms are adapted from the classical Euclidean algorithm. We will compare our newly proposed algorithms in this paper with the above algorithms.

DeWin et al. [Win et al. 1996] presented a fast software implementation of arithmetic operations in $GF(2^n)$. In their fast implementation, large finite fields are viewed as extensions of base field $GF(2^{16})$. In [Harper et al. 1992], similar algorithms with base field $GF(2^8)$ were developed. As the fast implementation was proposed before the algorithms of right-to-left comb method and its variants, DeWin et al. did not compare their performances. Additionally, the DeWin implementation was evaluated on a single finite field $GF(2^{176})$, and it is unknown how the presented performance results translate to other fields. Our paper tries to address these limitations, by evaluating the proposed algorithms in a more extensive way.

The previous work most relevant to our paper is by Greenan et al. [Greenan et al. 2007; 2008]. Greenan et al. [Greenan et al. 2007; 2008] described a variety of table lookup algorithms for multiplication and division operations over $GF(2^n)$, and evaluated their performance on several platforms. They concluded that the performance of different implementations of finite field arithmetic highly depends on the underlying hardware and workload. Our work differs from theirs in two aspects. First, their table lookup algorithms were implemented in small finite fields, up to $GF(2^{32})$. Second, they did not perform a comparison with the right-to-left comb method or its variants, currently the fastest known algorithms for multiplication. In our work, we study finite fields from $GF(2^{32})$ to $GF(2^{128})$, and we compare the performance of our algorithms with the left-to-right comb method with windows of width $w$.

For small finite fields, the result of an arithmetic operation could be directly looked up from pre-computed tables [Plank 1997]. The number of lookups depends on the table lookup algorithm. Huang et al. introduced a couple of efficient table lookup implementations [Huang and Xu 2003]. Their implementations do not contain any conditional branches and modular operations. This way greatly improves the performance of table lookup. Although their algorithms are designed for finite fields up to $GF(2^{16})$, they are also useful for implementing large finite fields. We incorporate some of their techniques in our proposed algorithms introduced in Section 4.

Besides targeted for a general platform, an implementation can be developed for a particular platform. Then, the implementation can take advantage of the instructions available at that platform to achieve high performance [Intel 2007; 2011]. For example, Aranha et al. [Aranha et al. 2010] introduced a new split form of finite field elements, and presented a constant-memory lookup-based multiplication strategy. Their approach made extensive use of parallel table lookup (PTLU) instructions to perform field operations in parallel. They have shown that their implementation is effective for finite fields from $GF(2^{113})$ to $GF(2^{1223})$ on the platforms supporting PTLU instructions. In this paper, we have a different focus. We aim to optimize the performance of a general software implementation for finite fields.

There are several open source implementations for finite fields. One implementation is `relic-toolkit` provided by Aranha et al. [Aranha 2010]. `relic-toolkit` is a cryptographic toolkit that emphasizes efficiency and flexibility. It provides a rich set of functionalities, including prime and binary field arithmetic. Another library is `Jerasure`, implemented by Plank [Plank et al. 2008]. `Jerasure` supports erasure coding in storage applications. It implements finite fields from $GF(2^4)$ to $GF(2^{32})$, but it does not support larger ones. Nevertheless, `Jerasure` provides a good framework for finite field arithmetic, and we utilize it to develop our code for larger finite fields.

## 3. ARITHMETIC OPERATIONS IN FINITE FIELDS

In this paper, motivated by our HAIL application [Bowers et al. 2009], we focus on arithmetic operations for large finite fields of characteristics two $GF(2^n)$, with $n = 16*m$, such as $GF(2^{64})$ or $GF(2^{128})$, where field elements can be byte aligned. Most techniques presented in this paper, however, can be readily applied to general finite fields $GF(2^n)$. In this section, we briefly introduce several well-known algorithms for arithmetic operations in finite fields, as well as their complexity analysis.

### 3.1 Binary Polynomial Method

According to finite field theory [Guajardo et al. 2006], elements of a finite field have multiple representations. In *standard basis* (or *polynomial basis*), an element of $GF(2^n)$ can be viewed as a polynomial $a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0$ of degree $n-1$ with coefficients in $GF(2)$. The same element can be represented with a bit vector $(a_{n-1}, a_{n-2} \ldots, a_1, a_0)$ of length $n$. To generate efficient machine representations, bit vectors are grouped into multiple machine words. For instance, in a 64-bit machine, a single *long* value holds an element of finite field $GF(2^{64})$. Elements of larger fields are represented with multiple long values, e.g., two *long* values are used for one element in $GF(2^{128})$.

There are other field representations, e.g., using a *normal basis* [Lidl and Niederreiter 1997]. A normal basis of $GF(2^n)$ is a basis of the form $(\beta, \beta^2, \ldots, \beta^{2^{n-1}})$, for some $\beta \in GF(2^n)$. An element in normal basis is represented as $b_{n-1}\beta^{2^{n-1}} +$

$b_{n-2}\beta^{2^{n-2}} + \dots b_1\beta^2 + b_0\beta$, where $b_i \in GF(2)$. The normal basis representation is efficient for speeding up exponentiations used in some cryptographic algorithms. In our paper, however, we focus on the standard basis representation.

In the standard basis representation, addition and subtraction in $GF(2^n)$ can be simply implemented using bitwise XORs of bit strings of length $n$. To implement multiplication and division, we need to consider first an *irreducible polynomial* $f(x)$ of degree $n$ over $GF(2)$ [Lidl and Niederreiter 1997]. Then multiplication and division are defined as follows:

**Multiplication of two polynomials $a(x)$ and $b(x)$:** A simple multiplication algorithm is the classical shift-and-add method [Schroeppel et al. 1995]. This method, however, is efficient in hardware, but not in software [Guajardo et al. 2006]. An efficient software implementation is the left-to-right comb method with windows of width $w$ [López and Dahab 2000]. This algorithm first multiplies $a(x)$ and $b(x)$, resulting in a polynomial of degree at most $2n-2$. Then, the multiplication result is reduced modulo the irreducible polynomial $f(x)$ to obtain a polynomial in $GF(2^n)$. More details on this method are provided below. Other similar methods include the right-to-left comb method and the left-to-right comb method [López and Dahab 2000], but these methods have been shown to be slower than the left-to-right comb method with windows of width $w$ [López and Dahab 2000].

We give now some details on the left-to-right comb method with windows of width $w$ for multiplication. This method computes the multiplication of two polynomials $a(x)$ and $b(x)$ of degree at most $n-1$ over $GF(2)$. It is intuitively based on the observation that if $b(x) \cdot x^k$ is computed for a $k \in [0, W-1]$, where $W$ is the machine word size, then $b(x) \cdot x^{Wj+k}$ can be computed by simply appending $j$ zero words to the right of $b(x) \cdot x^k$ ([Hankerson et al. 2000; López and Dahab 2000]). Furthermore, this method is accelerated significantly at the expense of a little storage overhead. It first computes $b(x) \cdot h(x)$ for all polynomials $h(x)$ of degree at most $w-1$, and then it can process $w$ bits of $a(x)$ at once rather than only one bit at a time. The pseudocode of this method is shown in Algorithm 1. In Algorithm 1, $a$, $b$, and $c$ are coefficient vectors representing polynomials $a(x)$, $b(x)$ and $c(x)$. $a$ is a vector of words of the form $(a[s-1], a[s-2], \cdots a[1], a[0])$, where $s = \lceil n/W \rceil$. Similar notations are used for $b$ and $c$. One thing to note is that as Algorithm 1 runs, the length of $c$ is $2s$, while the length of $a$ and $b$ is constant at $s$. More details of this method are available in [Hankerson et al. 2000; López and Dahab 2000].

In the multiplication operation, the left-to-right comb method with windows of width $w$ is followed by a modular reduction step in which the degree of $c(x)$ is reduced from at most $2n-2$ to at most $n-1$. Generally, modular reduction for a random irreducible polynomial $f(x)$ is performed bit by bit, i.e., the degree of $c(x)$ is reduced by one in each step. However, if $f(x)$ is a trinomial or pentanomial (i.e., it has three or five non-zero coefficients, recommended by NIST in the standards for public key cryptography [for Standards and Technology 2009]), the reduction step can be efficiently performed word by word [Guajardo et al. 2006]. Then, the degree of $c(x)$ is reduced by $W$ in one step, and the modular reduction of $c(x)$ is greatly sped up. In this paper, we only use trinomial or pentanomial irreducible polynomials for finite fields ranging from $GF(2^{32})$ to $GF(2^{128})$, and therefore we perform the modular reduction of the multiplication result one word at a time.

**Algorithm 1** Left-to-right comb method with windows of width $w$

```
INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most $n-1$ represented
    with vectors $a$ and $b$, and $s = \lceil n/W \rceil$
OUTPUT: $c(x) = a(x) \cdot b(x)$ represented with vector $c$
 1: Precompute $b_h = b(x) \cdot h(x)$ for all polynomials $h(x)$ of degree at most $w-1$
 2: $c \leftarrow 0$;
 3: for $k$ from $W/w - 1$ to 0 do
 4:    for $j$ from 0 to $s-1$ do
 5:       Let $h = (h_{w-1}, h_{w-2}, ..., h_1, h_0)$, where $h_t$ is bit $(wk+t)$ of $a[j]$
 6:       for $i$ from 0 to $s-1$ do
 7:          $c[i+j] \leftarrow b_h + c[i+j]$
 8:       end for
 9:    end for
10:    if $k \neq 0$ then
11:       $c \leftarrow c \cdot x^w$;
12:    end if
13: end for
```

**Division of two polynomials $a(x)$ and $b(x)$:** There are several different ways to implement the division operation. One method computes the inverse polynomial of $b(x)$ in $GF(2^n)$, denoted by $b^{-1}(x)$, and then multiplies $a(x)$ with $b^{-1}(x)$. Other methods directly compute the division result. Several of the popular division algorithms include the Extended Euclidean Algorithm, the Binary Extended Euclidean Algorithm and the Almost Inverse Algorithm [Hankerson et al. 2000; Schroeppel et al. 1995]. These algorithms are adapted from the classical Euclidean algorithm [Beachy and Blair 2006].

Efficient division algorithms, including the Extended Euclidean Algorithm, the Binary Extended Euclidean Algorithm, and the Almost Inverse Algorithm, are all based on the Euclidean algorithm. Here, we briefly describe the idea behind the Extended Euclidean Algorithm. Assume $f(x)$ is an irreducible polynomial of degree $n$ over $GF(2)$. For any $a(x)$ with coefficients in $GF(2)$, the Euclidean algorithm computes $\texttt{gcd}(a(x), f(x)) = 1$ (since $f(x)$ is irreducible). Then, according to algebra theory [Beachy and Blair 2006],

$$\exists\ b(x),\ c(x) \text{ s.t. } a(x) \cdot b(x) + f(x) \cdot c(x) = 1 \bmod f(x) \tag{1}$$

The Extended Euclidean Algorithm computes both $b(x)$ and $c(x)$ in equation (1) when calculating $\texttt{gcd}(a(x), f(x))$. It is easy to see that $a^{-1}(x) \bmod f(x) = b(x)$. Hence, the Extended Euclidean Algorithm computes $a^{-1}(x) \bmod f(x)$, the inverse of $a(x)$. Moreover, for fields of base 2, the Extended Euclidean Algorithm can be used to directly compute division without first obtaining $a^{-1}(x)$. The other two algorithms, the Binary Extended Euclidean Algorithm and the Almost Inverse Algorithm, are variants of the Extended Euclidean Algorithm optimized for $GF(2^n)$ [Guajardo et al. 2006].

As it is difficult to precisely analyze the time complexity of division, we instead provide the measured performance of division in Section 5. Interested readers can find the theoretical analysis of the Binary Extended Euclidean Algorithm in [Vallée

1998].

Using the standard basis representation, we implement the left-to-right comb method with windows of width $w$ for multiplication and the Binary Extended Euclidean Algorithm for division. We refer to the use of these algorithms for implementing finite field arithmetic as the *binary polynomial method.*

## 3.2 Table Lookup Methods

There are various table lookup methods that precompute and store results of arithmetic operations in tables with the goal of speeding up evaluation of multiplication and division operations. These methods achieve tradeoffs between the amount of storage for precomputed tables and operation speed.

### 3.2.1 *Full Multiplication and Division Tables.*

One simple table lookup method uses full multiplication and division tables. This algorithm precomputes the multiplication and division results for all element pairs in the field (by using, for instance, the binary polynomial method described in Section 3.1) and stores the results in tables. The tables are kept in main memory for small fields. To perform a multiplication or division operation, this algorithm quickly looks up the result from the tables with no computation.

While this algorithms involves only one table lookup for both multiplication and division, its space complexity is quadratic in the size of the field. For $GF(2^n)$, its storage complexity is $(n/8) * 2^{2n+1}$ bytes. For most off-the-shelf machines, this memory requirement is acceptable for $GF(2^8)$, but not for larger finite fields. For example, full multiplication and division tables for $GF(2^{16})$ would already need $2^{34}$ bytes, i.e., 16GB. Therefore, in this paper, we only use this table lookup algorithm for $GF(2^8)$.

### 3.2.2 *Log and Antilog Tables.*

Since all non-zero elements in a finite field form a cyclic group under multiplication [Beachy and Blair 2006], there exists a primitive element $\alpha$ in the field so that any non-zero element in the field is a power of the primitive element: for any $g \in GF(2^n)$, there exists an $0 \leq \ell < 2^n - 1$ such that $g = \alpha^\ell$. $\ell$ is called *the discrete logarithm* of element $g$ with respect to $\alpha$ in field $GF(2^n)$.

Based on this observation, a table lookup algorithm can be constructed [Plank 1997]. This algorithm builds two tables called log and antilog. The log table records the mapping from an element $g$ to its discrete logarithm $\ell$. Conversely, the antilog table records the mapping from power $\ell$ to a unique element $g$ in the field. These two tables can be built with the binary polynomial method for implementing exponentiation. After pre-computing these two tables, field operations can be performed as follows:

**Multiplication of two elements $g_1$ and $g_2$:** If $g_1$ or $g_2$ is 0, multiplication returns 0. Otherwise, do a lookup in the log table and get the discrete logarithms $\ell_1$ and $\ell_2$ for $g_1$ and $g_2$, respectively. Then, compute $\ell_3 = (\ell_1 + \ell_2) \bmod (2^n - 1)$. Finally, use the antilog table to find the field element $g_3$ corresponding to power $\ell_3$ and return $g_3$ as the multiplication result.

**Division of two elements $g_1$ and $g_2$:** If $g_1$ is 0, division returns result 0. Otherwise, use the log table to lookup the discrete logarithms $\ell_1$ and $\ell_2$ for $g_1$ and

$g_2$, respectively. Then, compute $\ell_3 = (\ell_1 - \ell_2) \bmod (2^n - 1)$. Finally, use the antilog table to find the field element $g_3$ corresponding to power $\ell_3$ and return $g_3$ as the division result.

Both multiplication and division involve four similar steps: (1) determine whether one element is 0; (2) perform a lookup in the log table; (3) compute the modular addition or subtraction; (4) use the antilog table to lookup the final result. Steps (1) and (3) could be optimized. Jerasure, for example, expands the storage of antilog tables by a factor of three to avoid step (3), which results in improved performance [Plank et al. 2008]. Huang et al. expand the antilog table by a factor of four to be able to remove both steps (1) and (3), and improve computation performance by up to 80% [Huang and Xu 2003].

In this paper, we make use of the optimizations in [Huang and Xu 2003] to implement this algorithm. The time complexity for both multiplication and division is then one addition (or subtraction) operation with three table lookups. The space complexity is $5 \cdot (n/8) \cdot 2^{n+1}$ bytes. Hence, this algorithm is applicable only to $GF(2^8)$ and $GF(2^{16})$ before memory demands become unreasonable.

## 3.3 Hybrid of Computational and Table Lookup Methods

The binary polynomial method evaluates the result of an arithmetic operation each time it is invoked. On the other hand, table lookup methods pre-compute and store all the results of arithmetic operations, resulting in very fast response time when an operation is invoked. In this section, we explore hybrid approaches that combine ideas from both methods to achieve computation efficiency for large finite fields with reasonable memory consumption.

3.3.1 *Split Tables.* The split table algorithm has been proposed by Huang and implemented in Jerasure by Plank [Plank 2007]. This algorithm is designed to optimize multiplication. To perform multiplication of two elements $g_1$ and $g_2$, this algorithm breaks each $n$-bit element in the field into $n/8$ units of size one byte. Then, it computes the result of multiplication by combining multiplication results of all unit pairs containing one byte from each operand. An example is shown below.

**Multiplication of two elements $g_1$ and $g_2$:** Suppose, for simplicity, that $g_1$ and $g_2$ are in $GF(2^{16})$. We represent $g_1$ as $[a_1, a_0]$, where $a_1$ is the high-order byte of $g_1$ and $a_0$ is the low-order byte of $g_1$. Similarly, we represent $g_2$ as $[b_1, b_0]$. By the distributive property of multiplication over finite fields, we can write:

$$\begin{aligned} g_1 * g_2 &= [a_1, a_0] * [b_1, b_0] \\ &= [a_1, 0] * [b_1, 0] + [a_1, 0] * [0, b_0] \\ &+ [0, a_0] * [b_1, 0] + [0, a_0] * [0, b_0] \end{aligned} \tag{2}$$

To perform the above multiplication efficiently, we can first use the binary polynomial method to build three multiplication tables called split tables [Plank 2007]. The tables store the multiplication results of all pairs of the form $[a_1, 0] * [b_1, 0]$, $[a_1, 0] * [0, b_0]$, and $[0, a_0] * [0, b_0]$. To evaluate $g_1 * g_2$, the results of multiplication for the four pairs in Equation (2) are looked up in split tables, and combined by bitwise XORs.

**Division:** This algorithm proceeds as in the binary polynomial method, which uses the Extended Euclidean Algorithm or its variants.

In general, for $GF(2^n)$, one multiplication needs $(n/8)^2$ table lookups. In terms of the space complexity, we need to build $n/4 - 1$ split tables for $GF(2^n)$, and the size of each table is $(n/8) * 2^{16}$ bytes. Thus, the total amount of storage needed is $(n/4-1) * (n/8) * 2^{16} = 2n(n-4)$ KB. For $GF(2^{64})$, this results in 7.5MB storage, an acceptable memory requirement. Therefore, this algorithm can be considered for large finite fields.

3.3.2 *Extension Field Method.* A more scalable algorithm to support large finite fields is the extension field method. This method makes use of precomputed tables in a smaller finite field, and several techniques for extending small field arithmetic into larger field operations.

**Extension field theory.** Section 3.1 describes the standard basis representation for elements of finite field $GF(2^n)$. In general, a finite field can use any of its proper base fields to represent its elements [Beachy and Blair 2006; Lidl and Niederreiter 1997]. For example, if $n = k \cdot m$, then field $GF(2^n)$ is isomorphic to $GF((2^k)^m)$. An element in $GF(2^n)$ can be represented as a polynomial $a_{m-1}x^{m-1} + a_{m-2}x^{m-2} \cdots + a_1x + a_0$ of degree $m-1$ with coefficients in $GF(2^k)$. We can use an irreducible polynomial of degree $m$ over $GF(2^k)$ to define the field arithmetic for $GF(2^n)$. With this representation, $GF(2^k)$ is named a *base field* of $GF(2^n)$, and $GF(2^n)$ an *extension field* of $GF(2^k)$.

For clarity, let us give an example for $GF(2^{16})$. If we consider it an extension field of $GF(2^8)$, then it becomes isomorphic to $GF((2^8)^2)$. We need to find two irreducible polynomials: one for the arithmetic in the base field $GF(2^8)$ (for instance $f(x) = x^8 + x^4 + x^3 + x^2 + 1$), and the second for generating the extension field $GF((2^8)^2)$ from base field $GF(2^8)$ (for instance $p(x) = x^2 + x + 32$).

**Multiplication of two elements $g_1$ and $g_2$:** Suppose that $g_1 = (a_1, a_0)$ and $g_2 = (b_1, b_0)$ are two elements in $GF((2^8)^2)$, with $a_0, a_1, b_0$ and $b_1$ in $GF(2^8)$, and $p(x)$ is an irreducible polynomial of degree 2 over $GF(2^8)$. Multiplication of $g_1$ and $g_2$ is performed as follows:

$$\begin{aligned} & (a_1x + a_0) * (b_1x + b_0) \\ = \; & (a_1 * b_1)x^2 + (a_1 * b_0 + a_0 * b_1)x + a_0 * b_0 \bmod p(x) \\ = \; & (a_1 * b_0 + a_0 * b_1 + 32 * a_1 * b_1)x \\ + \; & (a_0 * b_0 + 32 * a_1 * b_1) \end{aligned}$$

As all coefficients of $g_1$ and $g_2$ are from $GF(2^8)$, the multiplications and additions of coefficients in the above computation are performed in base field $GF(2^8)$. Addition is implemented as bitwise XOR, and multiplication in $GF(2^8)$ as table lookup.

For a general $GF(2^n)$, the time complexity of multiplication depends on the base field and the irreducible polynomial $p(x)$. One multiplication in the extension field $GF((2^k)^m)$ needs at least $m^2$ multiplications in the base field $GF(2^k)$. Let us give a justification for this theoretical lower bound. There are two steps involved in the multiplication of two elements in the extension field: multiplication of two polynomials of degree $m-1$ (resulting in $m^2$ multiplications in the base field), and reduction modulo the irreducible polynomial generating the extension field. If the irreducible polynomial used for generating the extension field has coefficients of only 0 or 1, no additional multiplications are needed in the second step. In practice, this bound may not be reachable since such an irreducible polynomial may not exist for some combinations of $GF(2^k)$ and $m$. More discussion on how to choose an irreducible polynomial $p(x)$ that reduces the number of multiplications in the base field is given in Section 4.

The space complexity for multiplication in extension field $GF(2^n)$ with $n = k \cdot m$ is exactly the same as that of base field $GF(2^k)$, and is thus independent from the extension field.

**Division of two elements $g_1$ and $g_2$:** Division in the extension field contains two steps: finding the inverse of $g_2$, and multiplying $g_1$ with $g_2^{-1}$. Computing the inverse of an element in the extension field can be implemented with the Extended Euclidean Algorithm. The Binary Extended Euclidean Algorithm and the Almost Inverse Algorithm used in the binary polynomial method are not applicable for extension fields.

## 4. EFFICIENT IMPLEMENTATION OF OPERATIONS IN EXTENSION FIELDS

In this section, we describe the main contribution of the paper, consisting of efficient implementation techniques for the extension field method.

### 4.1 Irreducible Polynomials

When implementing the extension field method, one important factor that impacts the performance of arithmetic operations is the choice of the irreducible polynomial used to construct the extension field. The irreducible polynomial determines the complexity of polynomial modular reduction and hence greatly affects multiplication and division performance. In general, there are multiple choices for irreducible polynomials, and our goal is to find those that optimize the performance of arithmetic operations.

4.1.1 *Impact of Irreducible Polynomials.* We give an example to demonstrate the great impact of irreducible polynomials on multiplication performance. Consider the extension field $GF((2^8)^4)$. When using $f(x) = x^8 + x^4 + x^3 + x^2 + 1$ as the irreducible polynomial over $GF(2)$ for $GF(2^8)$, there are two irreducible polynomials of degree 4 over $GF(2^8)$: $p_1(x) = x^4 + x^2 + 6x + 1$ and $p_2(x) = x^4 + 2x^2 + 5x + 3$. Either of them can be used to construct $GF((2^8)^4)$. The multiplication complexity in $GF((2^8)^4)$, however, is significantly different for these two irreducible polynomials and is shown in Table I.

| Irreducible polynomial | Multiplication | Addition |
|---|---|---|
| $x^4 + x^2 + 6x + 1$ | 16+3 | 18 |
| $x^4 + 2x^2 + 5x + 3$ | 16+9 | 18 |

Table I. Multiplication complexity in $GF((2^8)^4)$ when using two different irreducible polynomials.

In Table I, the second column shows the number of multiplications in the base field, and the third shows the number of additions in the base field. As multiplication is a much slower operation than addition, the number of multiplications in the base field dominates the performance.

In the second column, each number consists of two terms: the first term is the number of multiplications in the base field when multiplying two polynomials, and the second term is the number of multiplications performed when reducing the multiplication result modulo the irreducible polynomial. In our example, when multiplying two polynomials in $GF((2^8)^4)$, the cost of the first term is fixed, i.e., $4^2 = 16$ multiplications in $GF(2^8)$ (in this paper, we exclude the low probability of having the same multiplication pairs in the base field when performing a single multiplication for the extension field.) This number is independent of the irreducible polynomial $p(x)$ we are using. The cost of the second term, however, is determined by $p(x)$, and it can vary dramatically. In Table I, this cost is 3 multiplications for $p_1(x)$, but 9 multiplications for $p_2(x)$. Hence, the multiplication complexity for using $p_1(x)$ is 19 multiplications compared to 25 multiplications for $p_2(x)$, resulting in a 24% improvement in performance.

For larger finite fields, such as $GF(2^{128})$, the difference in performance between using a carefully chosen irreducible polynomial and a random one would be even more significant. Therefore, it is important to find efficient irreducible polynomials for optimizing performance.

4.1.2 *Test of Irreducible Polynomials.* There are many efficient irreducible polynomials over base field $GF(2)$ listed in the literature [Seroussi 1998]. However, for an arbitrary field, we have to search for good irreducible polynomials. During the search process, one key step is testing whether a polynomial is irreducible or not. A fast test algorithm is the Ben-Or algorithm [Ben-Or 1981; Gao and Panario 1997]. With the Ben-Or algorithm, we developed a test program using the NTL library [Shoup 1996]. Our experience shows that combing the Ben-Or algorithm with NTL leads to an efficient algorithm for testing polynomial irreducibility.

4.1.3 *Efficient Irreducible Polynomial Patterns.* Section 4.1.1 shows that the choice of irreducible polynomial greatly affects modular reduction efficiency. Particularly, it determines the number of multiplications in the base field. There is one key parameter of the irreducible polynomial that decides the number of multiplications performed in the base field: the number of coefficients not in $GF(2)$ (i.e., the number of coefficients different from 0 and 1, the only elements in $GF(2)$). We develop heuristics to search for irreducible polynomials that have the least number of coefficients not in $GF(2)$. In addition, we try to reduce the number of coefficients of 1 to decrease the number of additions during modular reduction.

We present some efficient irreducible polynomial patterns that we found through our search heuristics in Table II.

The 1st column of Table II is the value of $n$ in extension field $GF(2^n)$. The 2nd column is irreducible polynomials over base field $GF(2^8)$ for $GF(2^n)$, and the 3rd column is over base field $GF(2^{16})$. The irreducible polynomial used to construct $GF(2^8)$ is $f(x) = x^8+x^4+x^3+x^2+1$ over $GF(2)$, and it is $f(x) = x^{16}+x^{12}+x^3+x+1$ for $GF(2^{16})$.

| $n$ | $GF(2^8)$ | $GF(2^{16})$ |
|---|---|---|
| 32 | $x^4+x^2+6x+1$ | $x^2+x+8192$ |
| 48 | $x^6+x^2+x+32$ | $x^3+x+1$ |
| 64 | $x^8+x^3+x+9$ | $x^4+x^2+2x+1$ |
| 80 | $x^{10}+x^3+x+32$ | $x^5+x^2+1$ |
| 96 | $x^{12}+x^3+x+2$ | $x^6+x^3+8192$ |
| 112 | $x^{14}+x^3+x+33$ | $x^7+x+1$ |
| 128 | $x^{16}+x^3+x+6$ | $x^8+x^3+x+8$ |

Table II. Irreducible polynomials for extension fields $GF(2^n)$ over base field $GF(2^8)$ and $GF(2^{16})$

It can be proved that above irreducible polynomials are optimal in terms of the number of coefficients not in $GF(2)$. We consider two cases. 1) When constructing $GF(2^k)^m$, $k$ and $m$ are relative prime. For such $k$ and $m$, the presented irreducible polynomials in Table II only contain coefficients in $GF(2)$, so they are optimal. 2) $k$ and $m$ are not relative prime. A fact is that if a polynomial of degree $m$ that is irreducible over $GF(2)$, is also irreducible over $GF(2^k)$, then gcd$(m,k)$=1 [Lidl and Niederreiter 1997, ch. 3.3]. Thus, when $k$ and $m$ are not relative prime, equation gcd$(m,k)$=1 does not hold, and then any irreducible polynomial with degree $m$ over $GF(2^k)$ must have at least one coefficient not in $GF(2)$. In this case, because the presented irreducible polynomials in Table II contain only one coefficient not in $GF(2)$, they are also optimal.

With the above irreducible polynomials, one multiplication in $GF((2^k)^m)$ can be performed with $m^2$ or $m^2+m-1$ multiplications in base field $GF(2^k)$. If $k$ and $m$ are relative prime, the multiplication number is $m^2$; otherwise, it is $m^2+m-1$. As explained in Section 3.3.2, there are two steps involved in the multiplication of two elements in the extension field. The first is the multiplication of two polynomials of degree $m-1$ (resulting in $m^2$ multiplications in the base field), and the second is the modular reduction modulo the irreducible polynomial generating the extension field. If the irreducible polynomial only contains coefficients in $GF(2)$, the second step needs 0 multiplication; otherwise, if there is one coefficient not in $GF(2)$, the second step results in $m-1$ multiplications.

## 4.2 Multiplication Implementation

This section presents the multiplication implementation for the extension field method. For simplicity, we focus on the implementation for extension fields $GF((2^k)^m)$ where gcd$(m,k)\neq$1. The implementation for the simpler case where gcd$(m,k)$=1 can be easily derived.

In Section 4.1.1, we gave an example showing how the choice of the irreducible polynomial generating an extension field affects the efficiency of multiplication. Table I shows that with polynomial $p_1(x)$, we need to perform 19 multiplications in the base field $GF(2^8)$ for each multiplication in the extension field $GF((2^8)^4)$. If multiplication in the base field is implemented with full multiplication and division tables, this corresponds to 19 table lookups in the base field $GF(2^8)$ as one multiplication needs only one table lookup.

However, if log and antilog tables are used for multiplication in the base field, the number of lookups increases by a factor of three. This is because one multiplication now involves three table lookups, two to the log table and one to the antilog table. In this section, we provide an efficient multiplication algorithm for using log and antilog tables in the base field. The implementation greatly decreases the number of table lookups from $3(m^2+m-1)$ to $m^2+4m-1$ for fields of the form $GF((2^8)^m)$ and $GF((2^{16})^m)$. This is achieved by caching table lookup results and using them repeatedly.

The implementation contains two algorithms: the multiplication algorithm using log and antilog tables and the modular reduction algorithm specifically designed for the irreducible polynomials presented in Section 4.1.3. In the multiplication algorithm given in Algorithm 2, we multiply two polynomials $a(x)$ and $b(x)$ of degree at most $m-1$ with coefficients in the base field and output as a result a polynomial $c(x)$ of degree at most $2m-2$.

In Algorithm 2, $a$, $b$, and $c$ are coefficient vectors of polynomials $a(x)$, $b(x)$, and $c(x)$, respectively. Each element of these vectors represents a single coefficient in the base field. The variables *logtable* and *antilogtable* are lookup tables in the base field. They are built in advance.

**Algorithm 2** Multiplication using log and antilog tables

```
INPUT: Polynomials $a(x)$ and $b(x)$ of degree at most $m-1$
OUTPUT: Polynomial $c(x) = a(x) \cdot b(x)$ of degree at most $2m-2$
 1: $c \leftarrow 0$;
 2: for $k$ from 0 to $m-1$ do
 3:     $alog[k] = logtable[a[k]]$;
 4:     $blog[k] = logtable[b[k]]$;
 5: end for
 6: for $k_1$ from 0 to $m-1$ do
 7:     for $k_2$ from 0 to $m-1$ do
 8:         $c[k_1+k_2] \oplus= antilogtable[alog[k_1] + blog[k_2]]$;
 9:     end for
10: end for
```

As the output $c(x)$ of Algorithm 2 may have degree more than $m$, it has to be modularly reduced. Here, we provide an efficient modular reduction algorithm. Suppose the irreducible polynomial is of the form $p(x) = x^m + x^3 + x + v$. The powers $x^{2m-2}, \ldots, x^m$ can be reduced modulo $p(x)$ as follows:

$$\begin{aligned}
x^{2m-2} &\equiv (x^3+x+v)\cdot x^{m-2} \bmod p(x)\\
x^{2m-3} &\equiv (x^3+x+v)\cdot x^{m-3} \bmod p(x)\\
x^{2m-4} &\equiv (x^3+x+v)\cdot x^{m-4} \bmod p(x)\\
&\cdots\\
x^{m+1} &\equiv (x^3+x+v)\cdot x \bmod p(x)\\
x^{m} &\equiv x^3+x+v \bmod p(x)
\end{aligned}$$

Our developed reduction method is presented in Algorithm 3, which is similar to the modular reduction approach in [Win et al. 1996]. In Algorithm 3, $c$ and $d$ are coefficient vectors of input polynomial $c(x)$ of degree $2m-2$ and output polynomial

$d(x)$ of degree $m-1$, respectively. Similarly efficient algorithms could be given for other patterns of $p(x)$. Note that this algorithm reduces the degree of $c(x)$ by one each time the loop is executed (lines 2-6).

**Algorithm 3** Modular reduction

```
INPUT: Polynomial $c(x)$ of degree at most $2m-2$
OUTPUT: Polynomial $d(x) = c(x) \bmod p(x)$ of degree at most $m-1$
 1: $vlog$ = logtable[$v$];
 2: for $k$ from $2m-2$ to $m$ do
 3:     $c[k-(m-3)] \oplus = c[k]$;
 4:     $c[k-(m-1)] \oplus = c[k]$;
 5:     $c[k-m] \oplus = antilogtable[logtable[c[k]] + vlog]$;
 6: end for
 7: $d \leftarrow 0$;
 8: for $k$ from 0 to $m-1$ do
 9:     $d[k] = c[k]$;
10: end for
```

We proceed to analyze the complexity of our multiplication method. In Algorithm 2, $2m$ table lookups are performed in lines 2-5, and $m^2$ table lookups are performed in lines 6-10. In Algorithm 3, $2m-1$ table lookups are performed in lines 1-6. Adding all operations in Algorithms 2 and 3, we obtain the multiplication complexity: $m^2 + 4m - 1$ table lookups.

Similar multiplication and modular reduction algorithms can be derived by using full multiplication and division tables to implement operations in the base field. The corresponding time complexity is $m^2 + m - 1$ table lookups.

## 5. PERFORMANCE EVALUATION

In this section we evaluate the algorithms described above for large finite fields ranging from $GF(2^{32})$ to $GF(2^{128})$. We present performance results for different multiplication and division algorithms within a field. Section 6 describes a cloud storage application which utilizes erasure coding over such large fields. We evaluate its performance improvement from the use of our newly implemented algorithms compared to previous implementations.

### 5.1 Experiment Setup

5.1.1 *Platforms.* The multiplication and division tests were run on a variety of platforms in order to observe how their performances vary on different processors. We tested our implementations on four platforms, all using Intel 64-bit processors, spanning their current offering from low to high-end. Table III details the specifications of each platform.

All tests were run on a 64-bit version of Linux. As a result, one `int` value represents one element in $GF(2^{32})$; one `long` value, i.e., a computer word, holds an element in $GF(2^{48})$ and $GF(2^{64})$; two `long` values represent an element in fields from $GF(2^{80})$ up to $GF(2^{128})$.

| Platform | CPU speed | L2 cache | Model |
|---|---|---|---|
| P4 | 3.0GHz | 2MB | Pentium 4 |
| Pd | 2.8GHz | $2 \cdot 1$MB | Dual Core (D820) |
| Pc2d | 2.1GHz | 3MB | Core 2 Duo (T8100) |
| Pc2q | 2.4GHz | $2 \cdot 4$MB | Core 2 Quad (Q6600) |

Table III. Platforms under test.

5.1.2 *Implementations.* We evaluated five implementations listed in Table IV, representing three distinct methods. Throughout the rest of the paper, we simply use the names in the first column of Table IV to refer to various implementations. *binary* is based on the binary polynomial method from Section 3.1; specifically, it uses the left-to-right comb method with windows of width $w$ ($w = 4$) [López and Dahab 2000] for multiplication and the Binary Extended Euclidean Algorithm [Menezes et al. 1997] for division. Similar to [López and Dahab 2000; Avanzi and Thériault 2007; Aranha 2010], we chose width $w = 4$, which we expect provides optimum performance. *split* uses the split table method for multiplication from Section 3.3.1 and the same division algorithm as *binary*. *gf8 (full)*, *gf8 (log)* and *gf16 (log)* are based on the extension field method from Section 3.3.2. *gf8 (full)* uses base field $GF(2^8)$, with arithmetic operations based on full multiplication and division tables. *gf8 (log)* also uses base field $GF(2^8)$, but it implements arithmetic operations in $GF(2^8)$ by log and antilog tables. *gf16 (log)* is based on $GF(2^{16})$ with operations implemented using log and antilog tables.

| Implementation | Method |
|---|---|
| *binary* | binary polynomial |
| *split* | split table |
| *gf8 (full)* | extension field |
| *gf8 (log)* | extension field |
| *gf16 (log)* | extension field |

Table IV. Evaluated implementations for $GF(2^n)$ .

We developed all implementations for finite fields of interest, and borrowed the implementations of arithmetic operations in $GF(2^8)$ and $GF(2^{16})$ from Jerasure. The code is written in C and compiled using **gcc** with the **-O2** optimization flag, which is recommended for most applications [gentoo wiki 2010]. The code is single-threaded, and thus does not take advantage of multiple cores when present.

In addition to compiler optimizations, many manual optimizations are applied to each individual implementation for best performance. One common optimization is that we do not use one general multiplication or division function for all tested finite fields, but instead develop specific implementations for each field. This allows us to determine the size of data structures at compile time, rather than runtime, which improves performance significantly. Another two important optimizations are performed in the implementation of the left-to-right comb method with windows of width $w$ ($w = 4$) for implementation *binary*. First, in Algorithm 1, we manually unroll the loop from line 6 to line 8. Second, line 11 is actually an iteration. We also manually unroll this loop. We found that these two optimizations greatly improve multiplication performance. For instance, we achieve an improvement of 20% for $GF(2^{96})$ and 35% for $GF(2^{128})$ on platform Pc2q.

## 5.2 Comparison of All Implementations Using Table Lookups

This section compares the performance of all implementations heavily using table lookups. Regarding these implementations, table lookup is a dominant performance factor, so we first present table lookup numbers of each implementation in Table V. The table lists the number of table lookups needed for one multiplication (column 2). Table V shows that *gf16 (log)* performs the least number of table lookup operations and is followed by *split*. *gf8 (full)* and *gf8 (log)* need more table lookups than the previous two. It is worth noting that for implementation *gf16 (log)*, if `gcd`$(n/16, 16) \neq 1$, the table lookup number is $n^2/256 + 4n/16 - 1$; otherwise, it is $n^2/256 + 2n/16$.

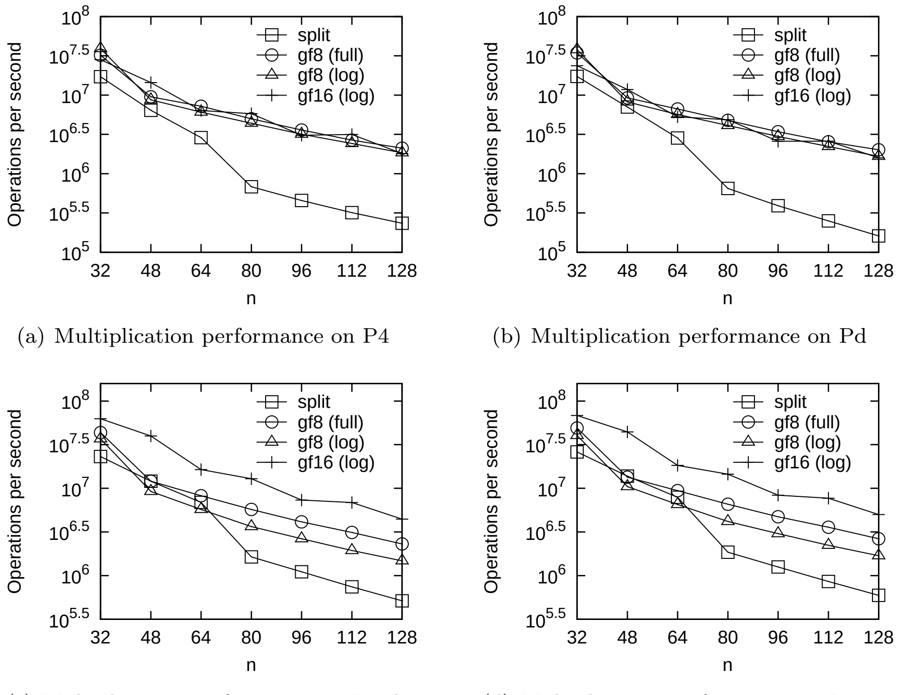
The space complexity of each implementation is given in the 3rd column of Table V. The size listed here is the combined space needed for both the multiplication and division algorithms. Note that the implementations based on the extension field method (*gf8 (full)*, *gf8 (log)*, and *gf16 (log)*) all use one `int` value to represent an element in $GF(2^8)$ or $GF(2^{16})$, which over-estimates the minimum space requirement, but leads to faster arithmetic operations. The table shows that the memory requirements for implementations based on the extension field method are independent of the size of the finite field. However, *split* consumes memory quadratic in $n$, limiting its practicality for large finite fields.

| Implementation | Table lookups | Memory needed |
|---|---|---|
| *split* | $n^2/64$ | $2n(n-4)$ KB |
| *gf8 (full)* | $n^2/64 + n/8 - 1$ | 0.5 MB |
| *gf8 (log)* | $n^2/64 + 4n/8 - 1$ | 5 KB |
| *gf16 (log)* | $n^2/256 + 4n/16 - 1$ (or $n^2/256 + 2n/16$) | 1.25 MB |

Table V. Table lookup number and memory needed of various implementations.

We now compare the measured performance of all these implementations. To measure the raw performance of an implementation, we randomly pick 36,000,000 element pairs from a finite field. The values of all element pairs are generated randomly. We then either multiply the pair, or divide one element by the other, and measure the elapsed time through the use of the `gettimeofday()` system call. Finally, we calculated how many operations are performed per second (operations per second). Each experiment is run 30 times and the average result is plotted in the graphs that follow. When the confidence level is set at 95%, the margin of error to the average value is less than 5% for all data points. As the margin of error is so small, we do not display error bars in the following figures.

5.2.1 *Multiplication.* Figure 1 displays the absolute multiplication performance on all four platforms. In the figure, the X-axis is the value of $n$. The Y-axis is the



(a) Multiplication performance on P4 (b) Multiplication performance on Pd

(c) Multiplication performance on Pc2d (d) Multiplication performance on Pc2q

Fig. 1. Multiplication performance of $GF(2^n)$ on various platforms.

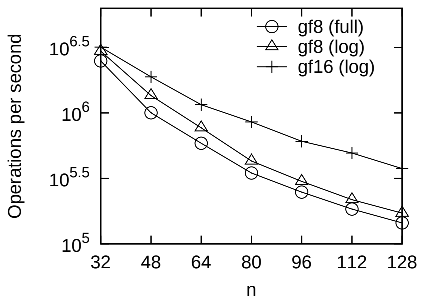
number of multiplications performed in one second, in base-10 log scale. Below are some observations that can be drawn from the data.

(1) Among table lookup intensive implementations, *gf16 (log)* outperforms all other implementations in most cases. The reason is that *gf16 (log)* performs about a quarter of the table lookups compared to other implementations, as shown in Table V. The performance gap is not significant in platform P4 and Pd due to their small L2 caches.

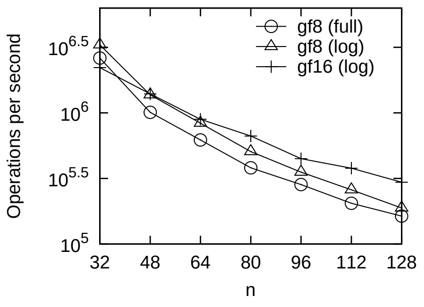
(2) *gf8 (full)* performs better than *gf8 (log)*, but the difference depends on platforms. *gf8 (full)* needs less table lookups than *gf8 (log)*, but its greater memory needed result in higher CPU cache miss ratio on platforms with small CPU cache, and thus these two implementations achieve similar performance on platforms P4 and Pd. However, on other two platform Pc2d and Pc2q, *gf8 (full)* outperforms *gf8 (log)* due to their large CPU caches.

(3) *split* has the worst performance of all the implementations in most cases. Although it uses a similar number of table lookups as *gf8 (full)* and *gf8 (log)*, it uses memory quadratic in $n$. This causes a large number of cache misses in larger fields, resulting in much worse performance.

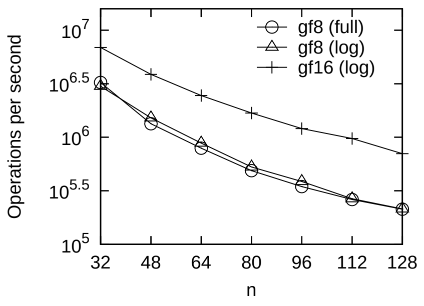
(4) As the size of our finite fields grows, the absolute performance of all table lookup intensive implementations decreases due to the increasing number of table lookups. All these implementations heavily depend on table lookups, and



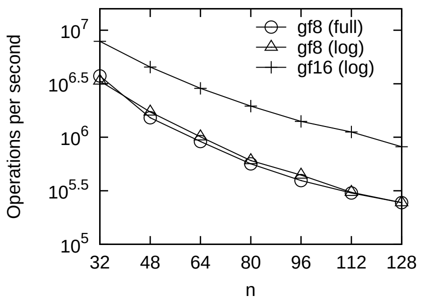
(a) Division performance on P4



(b) Division performance on Pd



(c) Division performance on Pc2d



(d) Division performance on Pc2q

Fig. 2. Division performance of $GF(2^n)$ on various platforms.

thus, their performance degrades almost linearly as the number of table lookups increases.

5.2.2 *Division.* Figure 2 displays the absolute division performance for all implementations except *split*. We omit *split* here as it uses the same division algorithm as *binary*. We observe the following:

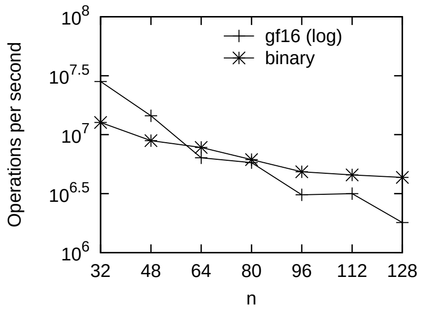
(1) *gf16 (log)* performs best among table lookup intensive implementations. As with multiplication, the performance advantage is highlighted on platforms with larger L2 caches.
(2) *gf8 (full)* performs worse than *gf8 (log)* on platforms with small L2 caches, but they perform similarly on platforms with large L2 caches.
(3) As with multiplication, as finite field size increases, the division performance of table lookup intensive implementations also decreases. However, the rate of decrease in division is slower than that of multiplication. For example, in platform Pc2q, the division performance of *gf16 (log)* on finite field $GF(2^{32})$ is 7% of its performance on finite field $GF(2^{128})$, while this number of multiplication performance is 10%.

5.2.3 *Summary.* The above results show that, among table lookup intensive implementations, *gf16 (log)* performs best and in most cases, by a large margin. This observation is consistent with the analysis shown in Table V, which indicates that *gf16 (log)* performs about a quarter of the table lookups and bitwise operations compared to other multiplication implementations. The memory requirements for *split* cause it to perform the worst, while the performances of *gf8 (full)* and *gf8 (log)* are between that of *split* and *gf16 (log)*.

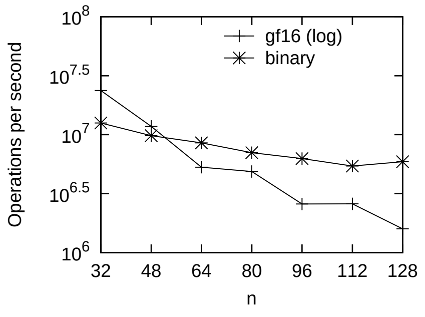
## 5.3 Comparison of *binary* and *gf16 (log)*

This section focuses on comparing the multiplication and division performance of *binary* and *gf16 (log)*, the best one among table lookup implementations.

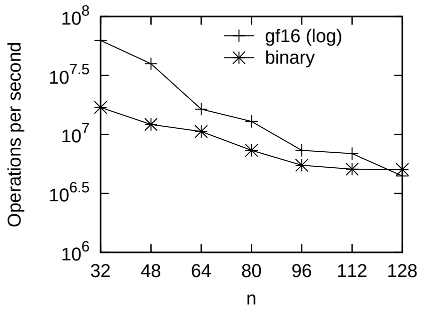
5.3.1 *Multiplication.* In Figure 3, graphs 3(a) - 3(d) display the multiplication performance comparison of *binary* and *gf16 (log)*. We have the following observations:



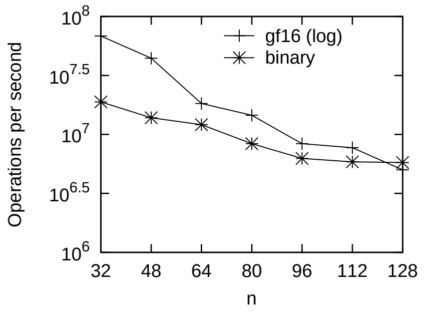
(a) Multiplication performance on P4



(b) Multiplication performance on Pd



(c) Multiplication performance on Pc2d



(d) Multiplication performance on Pc2q

Fig. 3. Multiplication performance of *binary* and *gf16 (log)* on various platforms.

(1) *gf16 (log)* performs better than *binary* from $GF(2^{32})$ to $GF(2^{48})$ on platforms P4 and Pd which only have at most 2MB L2 caches; on platforms Pc2d and Pc2q that have at least a 3MB L2 cache, *gf16 (log)* performs better from $GF(2^{32})$ to $GF(2^{112})$. For example, on Pc2q, the performance of *gf16 (log)* is higher than that of *binary* by 200% for field $GF(2^{32})$ and 35% for field $GF(2^{112})$.

(2) *gf16 (log)* outperforms *binary* for finite field $GF(2^{32})$ in all platforms, but as the finite field size grows, the performance gap between *gf16 (log)* and *binary* gradually decreases, with *binary* eventually outperforming *gf16 (log)*. This

transition occurs in all graphs. We could explain this effect using the analysis in Table V. The performance of *gf16 (log)* is dominated by table lookups, which is quadratic in the value of $n$, and thus its performance greatly decreases when $n$ grows. The *binary* implementation, however, is a computation intensive implementation and thus degrades less significantly.

(3) *gf16 (log)* starts performing slower than *binary* from finite field $GF(2^{64})$ on platforms P4 and Pd, while the starting field is $GF(2^{128})$ on platforms Pc2d and Pc2q. This is because P2d and Pc2q contains much larger CPU caches than P4 and Pd, and large cache size improves the performance of *gf16 (log)*, a table lookup intensive implementation. As L2 caches are currently increasing in size faster than CPU speed, *gf16 (log)* has the potential to surpass *binary*, for larger finite fields, in the near future. Nonetheless, the performance trend of *binary* makes it the best choice for extremely large finite fields for the foreseeable future.

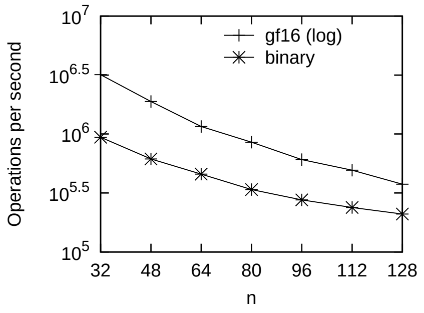
5.3.2 *Division.* Graphs 4(a) - 4(d) display the division performance comparison. The figure shows that in all cases, *gf16 (log)* greatly outperforms *binary.* For example, on platform Pc2q, the performance of *gf16 (log)* is 100% higher than that of *binary* for field $GF(2^{128})$ and 300% for field $GF(2^{32})$. Similar performance improvement can be observed on all other platforms. The reason is that the division algorithm of *binary* i.e., Binary Extended Euclidean Algorithm, works on one bit at a time since its base field is $GF(2)$. The division algorithm of *gf16 (log)*, however, works on 16 bits at a time due to its base field being $GF(2^{16})$. Hence, *gf16 (log)* is much more efficient than *binary* on division.

5.3.3 *Throughput Comparison.* To better understand the performance of finite fields from application perspective, we compared the performance of *gf16 (log)* with *binary* by throughput, i.e., how much data can be processed per second. As throughput is visible to applications, the performance comparison would be more useful in practice. The multiplication/division throughput is calculated as:

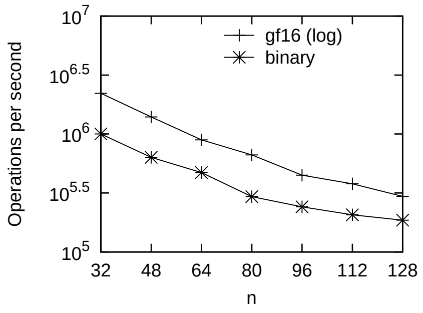
$$\text{Throughput} = \frac{\text{Operations per second * n}}{8} \tag{3}$$

This is derived as follows. Let $O$ be Operations per second. Then, $On$ is how many bits processed per second for multiplication or division, and $\frac{On}{8}$ is bytes processed per second, or Equation 3 for throughput.

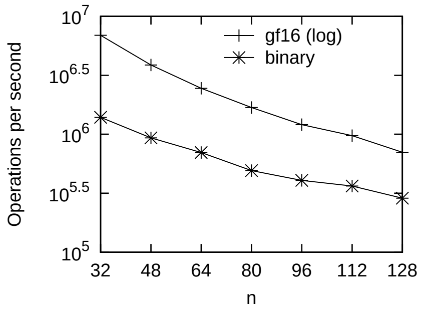
The throughput comparison is presented in Figure 5. Here, we only show comparison results on platform Pc2d. Interested readers can easily do the same conversions for other platforms from Equation 3. As for each finite field $GF(2^n)$, $n$ is the same to *gf16 (log)* and *binary*, operations per second determines the throughput of the implementations. Hence, Figure 5(a) shows the same performance comparison results of the two implementations as Figure 3(c). This also applies to the division performance shown in Figure 5(b) and 4(c). However, Figure 5(a) and 5(b) display different performance trends from their counterparts. In both figures, *gf16 (log)* degrades as $n$ grows, but *binary* keeps almost constant for all $n$. Because *gf16 (log)* achieves much higher performance than *binary* for $GF(2^{32})$, *gf16 (log)* can keep its performance advantage over $GF(2^{32})$ for a wide range of $n$ values. But for multiplication, *binary* performs better when $n$ starts from 128.



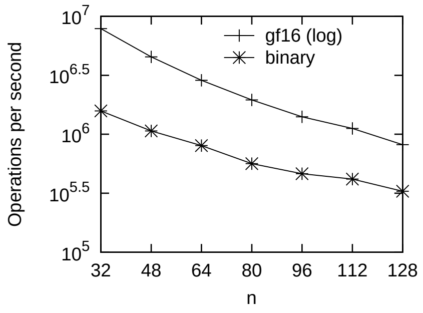
(a) Division performance on P4



(b) Division performance on Pd

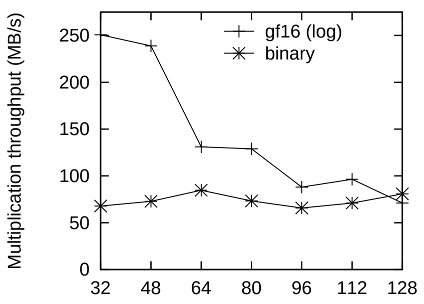


(c) Division performance on Pc2d

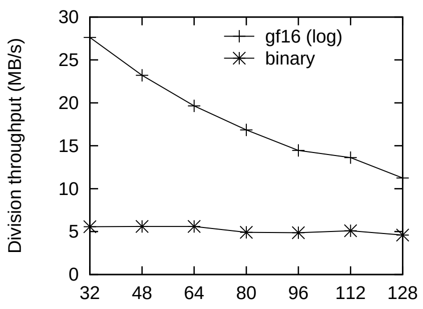


(d) Division performance on Pc2q

Fig. 4. Division performance of *binary* and *gf16 (log)* on various platforms.



(a) Multiplication performance

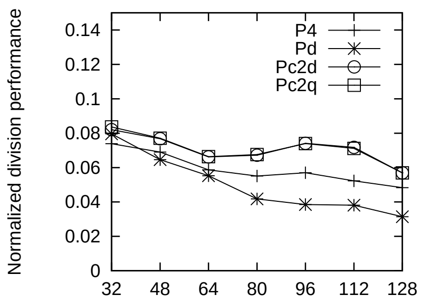


(b) Division performance

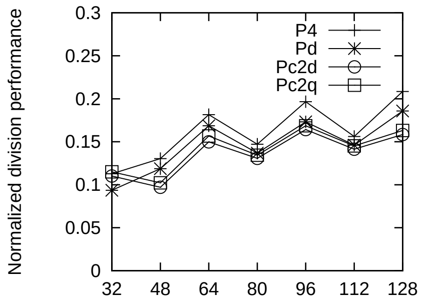
Fig. 5. Throughput comparison on platform Pc2d.

5.3.4 *Comparison of Division and Multiplication.* Because of the use of the Euclidean algorithm and its variants for implementing division, it is difficult to analyze the exact theoretical complexity of division. Here we focus on comparing the measured performance of division relative to that of multiplication. Figure 6 shows the normalized division performance for *binary* and *gf16 (log)*, computed as the division throughput divided by the multiplication throughput for each finite field.

We make the following observations:



(a) *binary* implementation



(b) *gf16 (log)* implementation

Fig. 6. Normalized division performance on various platforms.

(1) For *binary*, the normalized division performance decreases from about 0.08 to 0.03 across all test platforms. This is because the Binary Extended Euclidean Algorithm used in *binary* contains several conditional branches and iterations, and these greatly affect the division efficiency. The left-to-right comb method with windows of width $w$ ($w = 4$), however, has few conditional branches and only one iteration, allowing it to scale better as field size increases. This effect can be observed in Figure 3 too, which shows that the multiplication performance of *binary* degrades slower than that of division on all platforms.

(2) For *gf16 (log)*, the normalized division performance is fairly constant at about 0.15, regardless of finite field size. As table lookup operations dominate the multiplication and division performance, this result suggests that the table lookup complexity for division is $\Theta(n^2)$.

5.3.5 *Comparisons with Existing Implementations.* We compared the multiplication performance (in operations per second) observed from our implementations with the performance reported in existing literature. The comparison results are presented in Table VI.

| $n$ | *gf16 (log)* | LD-lcomb(w) | $n$ | LD-lcomb(w) [Avanzi 2007] | LD-lcomb(w) [Aranha 2010] |
|---|---|---|---|---|---|
| 48 | 39,820,605 | 12,149,787 | 47 | 15,625,000 | 2,969,642 |
| 64 | 16,375,242 | 10,579,802 | 59 | 8,547,009 | 2,836,374 |
| 80 | 12,889,563 | 7,328,084 | 79 | 5,847,953 | 1,981,301 |
| 96 | 7,340,030 | 5,474,563 | 89 | 4,830,918 | 1,946,795 |
| 112 | 6,883,625 | 5,073,405 | 109 | 3,649,635 | 1,902,230 |
| 128 | 4,440,471 | 5,046,587 | 127 | 3,278,689 | 1,958,632 |

Table VI. Multiplication performance comparison with existing implementations for $GF(2^n)$

In Table VI, column 1 to column 3 shows the performance results from our experiments on platform Pc2d. The 1st column is the value of $n$ in finite field $GF(2^n)$.

The 2nd column and the 3nd column are the implementations performance of *gf16 (log)* and LD-lcomb(w), i.e., left-to-right comb method with windows of width $w$ ($w = 4$). The 5th column is the performance of LD-lcomb(w) reported in [Avanzi and Thériault 2007] for finite fields listed in 4th column. The 6th column is our measured performance of the open source library `relic-toolkit` [Aranha 2010], a cryptographic library with emphasis on both efficiency and flexibility [Aranha 2010]. The tested field sizes for this library are in column 4.

First we compare our performance results with those in [Avanzi and Thériault 2007]. Avanzi et al. reported the performance of finite fields from $GF(2^{43})$ to $GF(2^{283})$, but we only list here their results for finite fields whose sizes are close to the ones we tested. As the performance presented in [Avanzi and Thériault 2007] is given in timing units, we translate their results to operations per second for comparison. Table VI shows that our implementation of LD-lcomb(w) achieves much higher performance (column 3) than those reported in [Avanzi and Thériault 2007] (column 5) in most cases. While we employ a slightly faster platform (Core 2 Duo with 2.1 GHz CPU compared to Core 2 Duo with 1.83 GHz CPU used by Avanzi et al.), we believe that the improvement is mainly due to our optimizations.

We also compare the performance of our implemented LD-lcomb(w) with that developed in `relic-toolkit` [Aranha 2010]. `relic-toolkit` is a general library, which supports arithmetic operations for any finite field when its irreducible polynomial is specified. We collected the performance of `relic-toolkit` by running it on platform Pc2d. Although `relic-toolkit` provides a framework to measure performance of field operations, we did not use their framework. Instead, we used our test framework and measured the performance of their implementation of LD-lcomb(w). Column 3 of Table VI shows that our implementation of LD-lcomb(w) greatly outperforms `relic-toolkit` (column 6). By looking into the source code of `relic-toolkit`, we found that `relic-toolkit` uses a general multiplication function, named `fb_mul_lodah`, to perform multiplication for all finite field sizes. The generality of this solution unavoidably sacrifices performance as it is not optimized for specific field sizes. In contrast, in our implementation the multiplication operation is tailored for different size fields. This confirms that manual optimizations are still necessary to optimize performance, as observed in [Avanzi and Thériault 2007].

Again, Table VI shows that the performance of *gf16 (log)* (column 2) is much higher than that of LD-lcomb(w) (column 3) in finite fields with sizes between $GF(2^{48})$ and $GF(2^{112})$.

## 6. HAIL AND 64-BIT ARITHMETIC

In this section, we show how the HAIL distributed cloud storage protocol introduced by Bowers et al. [Bowers et al. 2009] can benefit from using 64-bit finite field operations. We start by giving an overview of HAIL, and a new cryptographic primitive (an *integrity-protected dispersal code*) that is used by HAIL to reduce the amount of storage overhead for integrity checking. We describe two methods of constructing such codes that offer 64-bit security. One is based on 32-bit finite field arithmetic implemented in the Jerasure library, and the other on 64-bit operations developed in this paper. Finally, we evaluate and compare the two methods using

HAIL as our testbed.

## 6.1 HAIL overview

HAIL (High Availability and Integrity Layer) is a distributed cloud storage protocol for static data offering high availability guarantees to its users. The design of HAIL has been introduced recently by Bowers et al. [Bowers et al. 2009].

6.1.1 *RAID in the cloud.* HAIL provides a new way of building reliable cloud storage out of unreliable components. In that regard, it extends the principle of Redundant Arrays of Independent Disks (RAID) [Patterson et al. 1988] to the cloud. Similar to how RAID builds reliable storage at low cost from inexpensive drives, HAIL combines multiple, cheap cloud storage providers into a more robust and cost effective cloud storage offering.

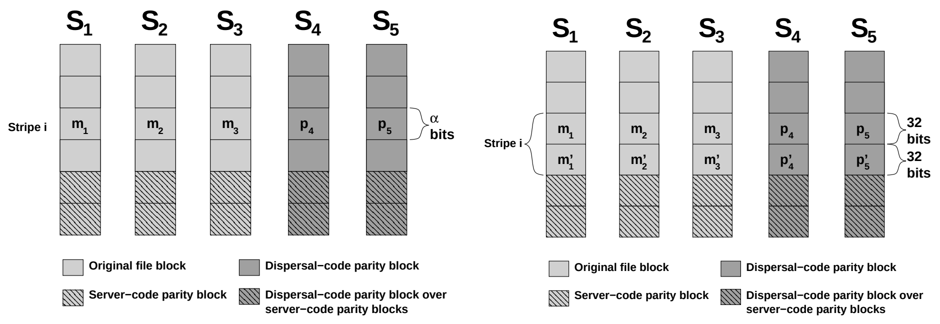
One of the main challenges in designing HAIL was to handle a strong adversarial model, and still retain the system efficiency in terms of storage overhead, computing costs and bandwidth requirements. While RAID has been designed to tolerate only benign failures (e.g., hard drive failures or latent sector errors in drives), we can not assume that cloud providers behave in a benign way. HAIL needs to deal with a strong, Byzantine adversarial model that models progressive corruption of providers over time. The adversarial model defined in [Bowers et al. 2009] is *mobile* in the sense that the adversary is allowed to corrupt $b$ out of $n$ providers in an epoch (an epoch is a time interval of fixed length, e.g. a week or a month).

6.1.2 *HAIL protocols.* HAIL includes the following protocols:

(1) File Encoding: Invoked by a client, this protocol distributes a file across $n$ providers and adds sufficient redundancy to enable recovery from provider failures.
(2) File Decoding: Reconstructs the file from a sufficient number of correct file fragments.
(3) Challenge-Response: Invoked periodically by the client, this protocol checks the availability of the file. The client contacts all providers, but needs only a threshold of responses in order to determine the status of the file.
(4) Fragment Reconstruction: Invoked when the challenge-response protocol detects corruption in at least one of the providers, this protocol recovers the corrupted fragments using the encoded redundancy.

6.1.3 *HAIL encoding.* A file $F$ is divided into fixed-size blocks, viewed as elements in a finite field $GF(2^{\alpha})$. The file is dispersed by the client across providers using a systematic $(n, \ell)$ erasure code [MacWilliams and Sloane 1977]. Striping is used for encoding efficiency: a stripe consists of $\ell$ file blocks and $n-\ell$ parity blocks, as depicted in Figure 7(a).

In addition, integrity checks are needed for file blocks to guarantee data integrity. HAIL introduces a new cryptographic primitive to reduce the amount of storage overhead for integrity checking, called an *integrity-protected dispersal code.* The main intuition for this construction is that message authentication codes (MACs) are embedded into the parity blocks of the dispersal code, and thus, do not use additional storage.



(a) Generic encoding for file blocks in $GF(2^{\alpha})$ (b) 64-bit encoding combining two 32-bit rows

Fig. 7. HAIL encoding.

As Figure 7 shows, HAIL makes use of an additional layer of encoding besides the dispersal code. Each file fragment stored at a provider after dispersal is encoded a second time with a server code. The role of the server code is to correct a small amount of corruption (which can not be detected through the challenge-response protocol). We can view the encoding of a file as a matrix with $n$ columns ($n$ is the total number of providers) and $|F|/\ell$ rows, with the dispersal code applied on rows, and the server code on columns.

6.1.4 *Integrity-protected dispersal code.* The integrity protected dispersal code is constructed, intuitively, from a Reed-Solomon erasure code and the addition of a unique random value per block computed from the file, row index, and column index. To generate a secure MAC, the parity symbols of the Reed-Solomon erasure code are obtained from the polynomial generated from message blocks evaluated at random points in the field.

## 6.2 Implementing a 64-bit integrity-protected dispersal code

The security level of the MACs embedded into the integrity-protected dispersal code depends on the finite field in which the Reed-Solomon encoding is performed. Most open source libraries implement encoding algorithms for symbols of length at most 32 bits. However, a 32-bit security level is weak from a cryptographic standpoint. We aim to obtain at least 64-bit security. In this section, we show two methods of achieving 64-bit security: the first using 32-bit arithmetic; and the second using the newly implemented 64-bit operations.

Assume that we divide the file into 64-bit blocks. As depicted in Figure 7(b), consider stripe $i$ consisting of data blocks $(m_1||m'_1, \ldots, m_\ell||m'_\ell)$ and parity blocks $(p_{\ell+1}||p'_{\ell+1}, \ldots, p_n||p'_n)$. Blocks $m_i, m'_i, p_j, p'_j$ are all in $GF(2^{32})$ for $i \in \{1, \ldots, \ell\}$ and $j \in \{\ell+1, \ldots, n\}$. Denote $\vec{m} = (m_1, \ldots, m_\ell)$ and $\vec{m'} = (m'_1, \ldots, m'_\ell)$.

Operating with polynomials over $GF(2^{32})$, we can obtain 64-bit security by considering the polynomial $f$ of degree $2\ell$ generated by $\vec{m}||\vec{m}'$. To obtain parity blocks $p_j$ and $p'_j$, we evaluate the polynomial $f$ at two different 32-bit random points.

Alternatively, if we operate with polynomials over $GF(2^{64})$, we can treat $(m_i||m'_i)$ as a single 64-bit value. To obtain parity block $p_j||p'_j$, for $j \in \{\ell+1, \ldots, n\}$, we evaluate the degree $\ell$ polynomial generated by $(m_1||m'_1, \ldots, m_\ell||m'_\ell)$ at a random

64-bit point.

## 6.3 Performance evaluation

We evaluate the performance of the two possible implementations of HAIL to achieve 64-bit security. The first uses the 32-bit arithmetic provided by Jerasure as described above, and the second our newly implemented 64-bit library. The two libraries use different algorithms and our goal is to assess the efficiency of our highly optimized implementation. We compare performance, specifically encoding and decoding throughput, across a number of parameters, including the size of the file, the number of servers over which the file was split, and the amount of parity information that was generated.

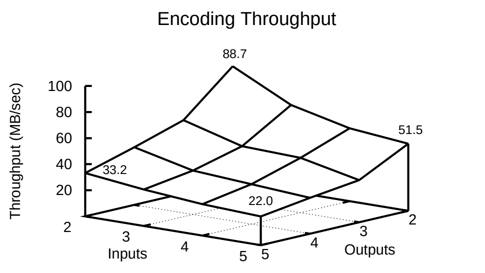
Tests were run on an Intel Xeon E5506, 2.13 GHz. quad-core processor which has a 4MB L2 cache. The system has 3GB of RAM available for processing and runs Red Hat Enterprise Linux WS v5.3 x86_64. All reported results are averages over 10 runs. Performance was measured during encoding of 400MB files, but reported numbers only indicate the throughput of the encoding/decoding functions, so are not affected by disk access time. Throughput is computed as the amount of data processed (400MBs), divided by the time spent performing the encoding/decoding function (clock time taken for the function call to return) after the data has been loaded into memory and properly formatted. As both encoding algorithms are single-threaded, performance is not improved by the additional cores.

Encoding performance can be drastically affected by a few key factors. These include the amount of data processed per call, or *packet size* (generally the more data per call the higher the throughput [Plank et al. 2009]), the encoding matrix (fewer ones leads to faster encoding), and obviously the amount of memory available. In our attempt to maximize performance we encode 512 rows of data per call (packet size = 8 bytes * 512 rows = 4096 bytes). Consistent with the application's usage, each encoding matrix is randomly generated and no optimizations are done to improve it.

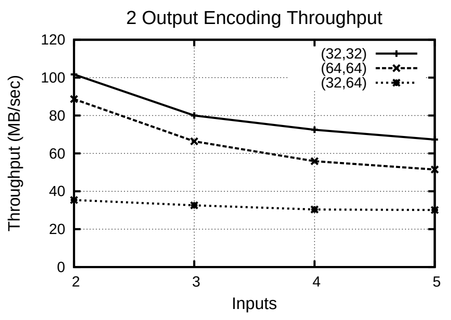
6.3.1 *Encoding Performance.* We first show in Figure 8(a) how the throughput of the 64-bit HAIL encoding using the new arithmetic library (denoted (64,64) HAIL) is dependent on both the number of inputs to the erasure code, as well as the number of outputs to be generated. The encoding throughput varies between 22 and 88 MB per second. We notice that the number of outputs has a slightly larger effect on performance than the number of inputs.

As we have shown, it is possible to achieve 64-bit security using a 32-bit encoding function. We denote the version of HAIL using 32-bit Jerasure arithmetic to achieve 64-bit security as (32,64) HAIL. To achieve 64-bit security using elements in $GF(2^{32})$ requires that the encoding matrix be doubled in both the number of rows as well as the number of columns, making it four times larger. The tradeoff is that, typically, operations in $GF(2^{32})$ require much less computational effort than operations in $GF(2^{64})$.

We compare real-world performance of the two possible implementations of HAIL encoding. Figure 8(b) shows the throughput of both 64-bit security versions of HAIL encoding, as well as the throughput achieved by a 32-bit secure encoding using Jerasure. The 32-bit encoding giving 32-bit security (denoted (32,32) HAIL)



(a) Encoding throughput by inputs and outputs for (64,64) HAIL

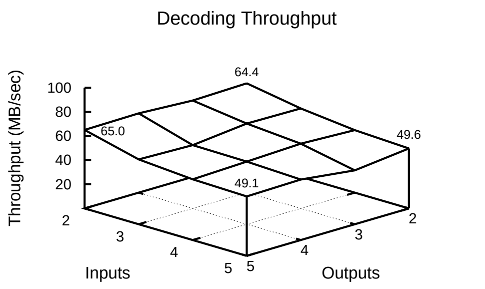


(b) Algorithmic differences in encoding throughput

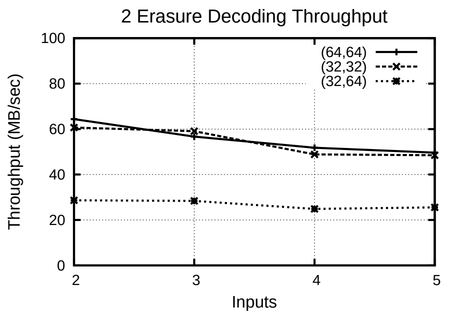
Fig. 8. Encoding throughput.

has the highest throughput, ranging from 70 to 100 MB per seconds, depending on the number of inputs. We are able to achieve 64-bit security with our newly implemented library ((64, 64) HAIL) with only about 10% reduction in throughput compared to the 32-bit Jerasure implementation ((32, 32) HAIL). In addition, we are able to double the encoding throughput compared to the version using the Jerasure 32-bit arithmetic operations to achieve 64-bit security ((32, 64) HAIL). We thus demonstrate the efficiency of our highly optimized arithmetic operations for large field arithmetic.

6.3.2 *Decoding Performance.* We show in Figure 9(a) the average decoding time to recover from two erasures for the (64,64) HAIL version. The decoding throughput is determined primarily by the number of erasures and the number of inputs to a lesser degree, and varies between 49 and 64 MB per second.



(a) Decoding throughput by inputs and outputs for (64,64) HAIL



(b) Algorithmic differences in decoding throughput

Fig. 9. Decoding throughput.

Again, it is also interesting to compare the effect of the newly developed 64-bit arithmetic library in HAIL with other implementations built on 32-bit Jerasure. Figure 9(b) shows the average decoding performance of (64,64) HAIL compared to the decoding throughput achieved by (32,64) HAIL and (32,32) HAIL. The optimizations in the newly developed 64-bit finite field operations achieve similar performance to Jerasure's native 32-bit decoding, at an increased security level. The decoding performance of the 64-bit new implementation is improved by a factor of two compared to (32,64) HAIL. This is due to the larger matrix necessary to achieve 64-bit security using 32-bit finite field operations, as well as the highly effective optimizations to arithmetic operations in $GF(2^{64})$ developed in this paper.

One thing the graph in Figure 9(a) hides is the effect of the location of erasures on decoding time. Since the erasure code used is systematic, errors in input (data) symbols require more work to recover from than errors in output (parity) symbols. This can be seen in Figure 10 where the effect of erasure location is demonstrated for the (64,64) HAIL decoding. For this graph, the encoding generates two outputs symbols, making the two erasures the maximum recoverable number of errors. Figure 10 shows the decoding throughput for the two erasures, comparing erasures in the outputs only (out,out), one each in the input and output (in,out), and erasures that only appear in the inputs (in,in). We notice that the decoding throughput can vary by more than a factor of two depending on erasure location.

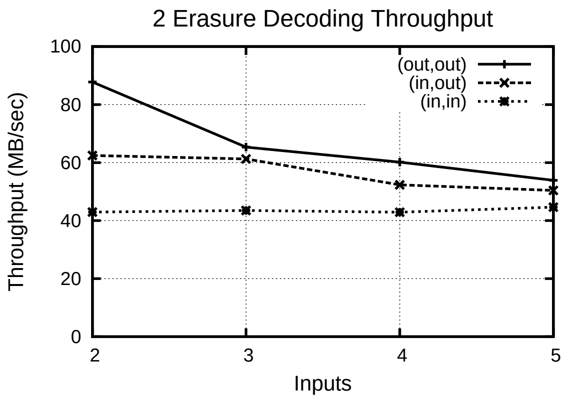


Fig. 10. Effect of erasure location on decoding throughput.

## 7. CONCLUSIONS

This paper provides new optimizations and efficient implementations of arithmetic operations for large finite fields $GF(2^n)$, ranging from $GF(2^{32})$ to $GF(2^{128})$ tailored for secure storage applications. We consider five different implementations based on three general methods for field arithmetic. We analyze the time and space complexity for these implementations, showing tradeoffs between the amount of computation and amount of memory space they employ. We also evaluate the raw performance of these implementations on four distinct hardware platforms, and present an application of our large field arithmetic implementation for distributed cloud storage.

Among the table lookup intensive implementations, we show that an implementation called *gf16 (log)*, based on the extension field method, achieves the best performance. The implementation uses precomputed log and antilog tables in $GF(2^{16})$ to speed up multiplication. We also compare the performance of *gf16 (log)* with that of the computation intensive implementation based on the binary polynomial method, called *binary*. The *binary* implementation uses the left-to-right comb method with windows of width $w$ for multiplication and the Binary Extended Euclidean Algorithm for division. We show that in platforms with small CPU cache, multiplication in *gf16 (log)* outperforms *binary* up to finite field $GF(2^{48})$. In platforms with large CPU cache, the range extends to $GF(2^{112})$. For division, *gf16 (log)* performs best in all cases. We conclude that *gf16 (log)* is an efficient implementation for large finite fields, particularly for modern CPUs with large CPU caches.

Finally, we show that our library is beneficial for HAIL, a distributed cloud storage protocol whose security relies on employing large field Reed-Solomon erasure coding. Our newly implemented 64-bit arithmetic operations speed up the encoding and decoding throughput of HAIL by a factor of two. We anticipate many secure storage applications requiring large finite field operations will directly benefit from our implementations presented in this paper. As future work, we plan to investigate further applications of large field arithmetic for erasure coding and cryptographic operations.

## REFERENCES

Aranha, D. F. 2010. RELIC is an Efficient Library for Cryptography, version 0.2.3. http://code.google.com/p/relic-toolkit/.

Aranha, D. F., López, J., and Hankerson, D. 2010. Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets. In *LATINCRYPT '10: The First International Conference on Cryptology and Information Security in Latin America*.

Avanzi, R. and Thériault, N. 2007. Effects of Optimizations for Software Implementations of Small Binary Field Arithmetic. In *WAIFI '07: International Workshop on the Arithmetic of Finite Fields*. 21–22.

Bailey, D. V. and Paar, C. 1998. Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms. In *CRYPTO '98: Proc. of the Annual International Cryptology Conference*.

Beachy, J. A. and Blair, W. D. 2006. *Abstract Algebra*. Waveland Press, Inc.

Ben-Or, M. 1981. Probabilistic Algorithms in Finite Fields. In *Symposium on Foundations of Computational Science*. 394–398.

Bowers, K., Juels, A., and Oprea, A. 2009. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In *CCS '09: Proc. of the 16th ACM Conference on Computer and Communications Security*.

Diffie, W. and Hellman, M. E. 1976. New Directions in Cryptography. *IEEE Transactions on Information Theory 22 (6)*, 644–654.

ElGamal, T. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory 31,* 4, 469–472.

for Standards, N. I. and Technology. 2009. FIPS 186-3: Digital Signature Standard (DSS). http://www.itl.nist.gov/fipspubs/by-num.htm.

Gao, S. and Panario, D. 1997. Tests and constructions of irreducible polynomials over finite fields. In *FoCM'97: Foundations of Computational Mathematics*.

gentoo wiki. 2010. http://en.gentoo-wiki.com/wiki/CFLAGS.

Greenan, K. M., Miller, E. L., and Schwarz, T. J. E. 2007. Analysis and Construction of Galois Fields for Efficient Storage Reliability. In *Technical Report UCSC-SSRC-07-09*.

GREENAN, K. M., MILLER, E. L., AND SCHWARZ, T. J. E. 2008. Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications. In *MASCOTS '08: International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems.*

GUAJARDO, J., KUMAR, S. S., PAAR, C., AND PELZL, J. 2006. Efficient Software-Implementation of Finite Fields with Applications to Cryptography. *Acta Applicandae Mathematicae 93*, 3–32.

HANKERSON, D., HERNANDEZ, J. L., AND MENEZES, A. 2000. Software Implementation of Elliptic Curve Cryptography Over Binary Fields. In *CHES '00: Workshop on Cryptographic Hardware and Embedded Systems.*

HARPER, G., MENEZES, A., AND VANSTONE, S. 1992. Public-Key Cryptosystems with Very Small Key Lengths. In *Eurocrypt '92: Proc. of the Annual International Conference on the Theory and Applications of Cryptographic Techniques.*

HUANG, C. AND XU, L. 2003. Fast Software Implementation of Finite Field Operations. *Technical report, Washington University*.

INTEL. 2007. Intel SSE4 Programming Reference. http://software.intel.com/file/18187/.

INTEL. 2011. Intel Advanced Encryption Standard (AES) Instructions Set. http://software.intel.com/file/24917.

KRAVITZ, D. W. 1993. Digital signature algorithm. U.S. Patent 5,231,668.

LIDL, R. AND NIEDERREITER, H. 1997. *Finite Fields.* Cambridge University Press.

LÓPEZ, J. AND DAHAB, R. 2000. High-speed Software Multiplication in $\mathbb{F}_{2^m}$. In *INDOCRYPT '00: Proc. of the Annual International Conference on Cryptology in India.*

MACWILLIAMS, F. J. AND SLOANE, N. J. A. 1977. *The Theory of Error Correcting Codes.* Amsterdam: North-Holland.

MENEZES, A., VAN OORSCHOT, P., AND VANSTONE, S. 1997. *Handbook of Applied Cryptography.* CRC Press.

MILLER, V. S. 1986. Use of Elliptic Curves in Cryptography. In *CRYPTO 85': Proc. of the Annual International Cryptology Conference.*

PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88: Proc. of the 1988 ACM SIGMOD International Conference on Management of Data.*

PLANK, J. S. 1997. A Tutorial on Reed-Solomon Coding for Fault Tolerance in RAID-like Systems. *Software – Practice and Experience 27 (9)*, 995–1012.

PLANK, J. S. 2007. Fast Galois Field Arithmetic Library in C/C++. http://www.cs.utk.edu/~plank/plank/papers/CS-07-593/.

PLANK, J. S., LUO, J., SCHUMAN, C. D., XU, L., AND WILCOX-O'HEARN, Z. 2009. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage. In *FAST '09: Proc. of the 7th Usenix Conference on File and Storage Technologies.*

PLANK, J. S., SIMMERMAN, S., AND SCHUMAN, C. D. 2008. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications. Tech. Rep. CS-08-627, University of Tennessee. August.

REED, I. S. AND SOLOMON, G. 1960. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics 8 (10)*, 300–304.

SCHROEPPEL, R., ORMAN, H., MALLEY, S. O., AND SPATSCHECK, O. 1995. Fast Key Exchange with Elliptic Curve Systems. In *CRYPTO '95: Proc. of the Annual International Cryptology Conference.*

SEROUSSI, G. 1998. Table of Low-Weight Binary Irreducible Polynomials. http://www.hpl.hp.com/techreports/98/HPL-98-135.pdf.

SHOUP, V. 1996. A New Polynomial Factorization Algorithm and Its Implementation. *Journal of Symbolic Computation 20*, 363–397.

VALLÉE, B. 1998. The Complete Analysis of the Binary Euclidean Algorithm. In *ANTS '98: Proc. of the Third International Symposium on Algorithmic Number Theory Symposium.*

WIN, E. D., BOSSELAERS, A., VANDERBERGHE, S., GERSEM, P. D., AND VANDEWALLE, J. 1996. A Fast Software Implementation for Arithmetic Operations in $GF(2^n)$. In *ASIACRYPT '96:*

*Proc. of the Annual International Conference on the Theory and Application of Cryptology Information Security.*