

Alex Shaindlin

RECURSION STRATEGIES FOR FAST BITSLICED POLYNOMIAL MULTIPLICATION OVER BINARY FIELDS

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Nicola Taveri
Alejandro Cabrera Aldaya
October 2024

ABSTRACT

Alex Shaindlin: Recursion strategies for fast bitsliced polynomial multiplication over binary fields
Master of Science Thesis
Tampere University
Theoretical Computer Science
October 2024

Elliptic curve cryptography requires a fast and secure implementation of the multiplication operation in the underlying field. Since the field elements involved are not large enough for the asymptotically optimal multiplication algorithms to always perform best in practice, implementing fast multiplication on a particular computer architecture requires experimental testing to determine which algorithm (or combination of algorithms) performs best for a given field size. Another approach to speeding up computation in binary fields is bitslicing, where a batch of elements is transposed and then operated on bitwise using SIMD vector instructions. Combining a bitsliced base layer with automated generation and testing of different recursion strategies, we offer a way to generate highly efficient library code for different field sizes and different architectures without manual tuning.

The previously published paper *Batch Binary Weierstrass* (appendix A) gave an overview of the software tool developed for generating and benchmarking the strategies, which uses a subset of the multiplication algorithms presented by Bernstein [3]. This thesis describes the tool in more detail, as well as presenting the necessary mathematical background, including worked examples of the Karatsuba and Toom-Cook fast multiplication algorithms. It also includes a discussion of how the tool has changed since the original publication, since bugs discovered during the writing of this thesis necessitated a rewrite of the strategy generation algorithm (although the generated code itself for any given strategy remains unchanged).

The best-performing strategies are established to be competitive with or even superior to other existing implementations of binary field multiplication at cryptographically relevant field sizes. The structural properties of the entire collection of generated strategies are explored, expanding our understanding beyond previous work which had only focused on experimental performance measurement. Reducing the range of subproblem sizes permitted to serve as base cases for the recursion is shown to have only a small effect on the size of the search space. It is established that the number of generated strategies stays in a narrow constant range for field sizes up to approximately $GF(2^{465})$ but begins to increase rapidly after that point, suggesting that it would be fruitful to investigate the possibility of using benchmarking results from smaller field sizes to prune known-inefficient strategies for subproblems when generating strategies for larger field sizes. The cause of incorrect multiplication results produced by the generated code for some strategies is revealed to be a bug in one of the C macros, which only manifests when it is expanded with a particular combination of inputs.

Keywords: cryptography, fast multiplication, Galois fields, recursive algorithms

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

No “artificial intelligence” tools (large language models) were used in the writing or editing of this thesis.

PREFACE

This thesis was written in Tampere, Finland in 2022 and 2023, mostly building on work I had completed in 2019 but also including new contributions and discoveries which the writing process itself inspired and enabled me to make. It would not have been possible without my supervisors, Billy Bob Brumley and Nicola Tuveri, and my fellow student researchers Kide Vuojärvi and Sohaib ul Hassan. I am also endlessly grateful for the love and support of my parents Martha and Andy, all my wonderful friends, and most particularly my partner Lindsay, who was instrumental in helping me find my footing and return to my studies after two very difficult years.

Tampere, 27th October 2024

Alex Shaindin

CONTENTS

1.	Introduction	1
1.1	Problem overview	1
1.2	Solution overview.	2
1.3	Research scope and goals	3
1.4	Structure of the thesis	4
2.	Background.	5
2.1	Algebraic structures.	5
2.1.1	Groups	5
2.1.2	Rings.	9
2.1.3	Fields	10
2.2	Elliptic curves	12
2.3	Cryptographic applications	14
2.3.1	One-way functions	15
2.3.2	The integer factorization problem and RSA	16
2.3.3	The discrete logarithm problem and Diffie–Hellman–Merkle	17
3.	Polynomial multiplication.	20
3.1	Schoolbook multiplication	20
3.2	Karatsuba multiplication	21
3.3	Toom–Cook multiplication	22
3.3.1	Description of the algorithm	24
3.3.2	Example	26
3.3.3	Equivalence of Karatsuba and Toom-2	27
3.4	Comparison of algorithm performance bounds	28
4.	Bitslicing	30
4.1	Overview	30
4.2	Simulating data-dependent branching.	32
5.	The finite field arithmetic library	34
5.1	Overview	34
5.2	Strategy generation.	35
5.3	Code generation and benchmarking	39
5.4	History	40
6.	Results	42
7.	Conclusion	47
	References.	48

Appendix A: Batch Binary Weierstrass	52
Appendix B: Strategy counts	74

GLOSSARY

DHMKE	Diffie–Hellman–Merkle Key Exchange
IETF	Internet Engineering Task Force
ISA	Instruction Set Architecture
NIST	National Institute of Standards and Technology
\mathbb{Q}	Rational numbers
\mathbb{R}	Real numbers
SIMD	Single Instruction, Multiple Data
\mathbb{Z}	Integers
\mathbb{Z}_n	Integers modulo n

1. INTRODUCTION

For thousands of years, secret communication relied on both parties agreeing in advance on a shared key that would be used to encrypt and decrypt their communications. This could be tricky and error-prone: for example, ciphers with a short key are usually extremely vulnerable to cryptanalytic attacks (the Vigenère cipher remained unbroken for three centuries, but those who know the right technique can break it with paper and pencil in a single afternoon), while a one-time pad is perfectly cryptographically secure in theory but can only be used to encrypt messages shorter than the key length and requires both parties to keep their copy of the key secure and coordinate to maintain the same position in the pad. [9, Chapter 2] In the 1970s, the invention of public-key cryptography revolutionized the scene: Whitfield Diffie and Martin Hellman (drawing on the work of Ralph Merkle) published an algorithm that enabled two parties to agree on a secret encryption key across a public channel, and Rivest, Shamir, and Adleman published an algorithm for key agreement as well as encryption and decryption across a public channel, which came to be known as RSA.¹ Although public-key algorithms can also be used to encrypt and decrypt data, they are extremely slow at doing so when compared to symmetric (shared key) algorithms. Today, hybrid encryption schemes allow us to have the best of both worlds: the parties first devise a shared key using a public-key algorithm, and then use that key to encrypt their communications with a symmetric algorithm. [36, Section 6.2.1]

1.1 Problem overview

Public-key cryptosystems become harder to break as longer encryption keys are used, but computation with larger keys takes more time. Cryptographers are always searching for the sweet spot: how much performance can we squeeze out of our systems while still maintaining acceptable levels of security? Setting aside advances in the design and manufacturing of the physical hardware itself, there are two main avenues to explore.

One is the design of new cryptosystems that can provide equivalent security with shorter keys (or, from another point of view, better security with the same key length). Elliptic

¹It later emerged that both of these algorithms had already been developed in secret by British intelligence agencies, but hadn't been put to much use prior to their publication. [43]

curve cryptography, which was first proposed independently by Koblitz [27] and Miller [34] in the 1980s, is an attractive alternative to encryption schemes based on other types of algebraic structures (such as RSA, which is based on modular integer arithmetic) for this reason. [22, Section 1.3] The other is the design of new algorithms, or improvements to the implementations of existing ones, that increase the efficiency of the underlying operations used in an existing cryptosystem (such as multiplication, exponentiation, and bit permutation). Since the design of entirely new cryptosystems is an extraordinarily difficult task and the cryptography community is (justifiably!) slow to accept them, improving the implementation of existing ones is crucial for steady progress.

In cases where “the” algorithm used for a given operation is actually a family of closely related algorithms in which the exact sequence of steps for a particular input can depend on configuration choices made at compiletime or runtime, the performance of different configurations in different situations can be experimentally tested to choose ones that perform well in practice. Some operations can also be sped up with batch computations, where clever use of resources makes it possible to process k elements simultaneously and produce results for the entire batch in less time than it would have taken to run the most efficient algorithm for an individual element k separate times. This thesis applies both of these methods to improve the performance of an elliptic curve cryptosystem implementation.

1.2 Solution overview

For a long time, elliptic curves over prime fields were considered the most efficient in software, but recent advances have shown that elliptic curves over binary fields can be competitive and in some cases even faster. [3] Multiplication in a binary or prime field is much more computationally intensive than addition; therefore, efficient polynomial multiplication is a cornerstone of efficient elliptic curve cryptosystem implementation. Fortunately, because multiplication is a fundamental operation in its own right and also forms the backbone of many algorithms for other important tasks like factorization and primality testing, multiplication algorithms have been a lively research area for decades. In addition to the “schoolbook” method, in which two elements are multiplied together digit-by-digit (for numbers) or term-by-term (for polynomials), there exist several different recursive multiplication algorithms, where two elements are split into parts which are operated on separately, producing intermediate results which are recombined to produce the final result. At larger input sizes, recursive multiplication is significantly faster, but the exact cutoff points at which specific recursive algorithms begin to outpace each other are highly sensitive to implementation details. [19, Chapter 8]

When many inputs are available at once, batch computation can be performed using the technique known as “bitslicing”, increasing throughput. In bitsliced computation, rather

than storing one w -bit value in each register, the values are “sliced” (transposed) so that one register contains the first bit of each input, the next register contains the second bit of each input, and so on. [5, 6, 30, 10] This can pose implementation difficulties, since it requires that all values be operated on with the exact same sequence of steps, but the improvement in throughput can be significant if large enough batches of input are available at once.

Bernstein [3] described several closely related recursive multiplication algorithms for binary fields, building on the well-known Karatsuba and Toom algorithms (see chapter 3), and provided C code for bitsliced batch computation using those algorithms. This thesis builds on that work and develops a software tool, `gf2sliced`, for emitting code for the optimal combination of recursion choices for bitsliced batch multiplication of elements of a given binary field on a given Instruction Set Architecture (ISA). The series of steps chosen is called the “splitting strategy”, since it primarily concerns how to split the field elements into smaller subproblems when recursing. We benchmark the generated libraries for the best-performing splitting strategies on multiple architectures and compare its performance to other cryptographically relevant finite field arithmetic implementations, with favorable results. In addition, the splitting strategy generation algorithm itself has been rewritten to eliminate a major bug that had gone undetected in previous versions of the tool. The result is a command-line tool that handles the entire process of strategy generation, benchmarking, and compilation, to produce a customized binary field library with highly competitive performance without requiring manual tuning of the library code by an expert assembly language programmer.

1.3 Research scope and goals

The scope of the work is to automate the process of generating different multiplication strategies (using the C functions provided in [3]), benchmarking them, and compiling a version of the binary field arithmetic library that uses the best-performing strategy. The scope includes making bugfixes and usability/maintainability improvements to the implementation of our strategy generation algorithm, and to the build process for our tool. Making changes to the C functions themselves is out of scope, as is the use of specialized instructions for binary field arithmetic (e.g. ARMv8 `PMULL`) and the incorporation of additional recursive algorithms such as Schönhage-Strassen.

The original goal of the project was to demonstrate that the best strategies generated this way are competitive with other existing approaches in the literature. The experimental results obtained to that effect were published in [11], which is included here as Appendix A, and similar experiments have also been examined in more statistical detail in [42]. In addition to that, the goals of this thesis and the work documented therein are also:

- To present a detailed description of the strategy generation algorithm—for the first

time accurately representing the behavior of the software, since the previously published descriptions [11, 42] aspirationally reflected our intent but did not account for bugs that were then present in the implementation and have been fixed during the writing of this thesis.

- To analyze the properties of the entire collection of generated strategies (including testing how those properties change when a heuristic proposed in [42] is applied).
- To identify why the generated code for some generated strategies for some field sizes produces incorrect results. In particular, it is shown that there is exactly one split which is present in all of the buggy strategies and none of the correct ones, which means strategies that will produce buggy code can be preemptively eliminated at generation time.

The software tool itself is also finally made available as an artifact.

1.4 Structure of the thesis

Chapter 2 explains the mathematical background for the work, giving an overview of groups, rings, fields (with an emphasis on binary field arithmetic), and elliptic curves, describing some uses of these algebraic structures in cryptography both in the present day and historically. Chapter 3 discusses polynomial multiplication in more depth, examining some popular recursive algorithms and considering the tradeoffs between straightline and recursive multiplication. Chapter 4 introduces bitslicing, explaining how it works with bit lanes of varying sizes on dedicated Single Instruction, Multiple Data (SIMD) architectures as well as how any instruction set architecture can be used for 1-bit lane bitslicing regardless of native SIMD support, and how a carefully written SIMD implementation can provide resistance to certain types of side-channel attacks. Chapter 5 describes our contribution, going into detail on the strategies used for generating different possible implementations of the finite field arithmetic layer and for then benchmarking them to find the optimal one on a specific architecture. In chapter 6, the benchmarking results are presented and analyzed. Finally, chapter 7 provides a summary of the work and a discussion of future avenues of exploration.

2. BACKGROUND

2.1 Algebraic structures

This section assumes a familiarity with basic set theory, including maps (functions) and products (pairing). No particular axiomatic set theory is used; naive set theory is enough. The definitions in this section are largely due to Lidl and Niederreiter [29], with the addition of a multiplicative identity element for rings. Textbooks covering these subjects at an approachable level include Dummit and Foote [16] and Artin [2].

A *binary operation* on a set S is a mapping $S \times S \rightarrow S$; that is, to each pair $(q, r) \in S \times S$, it associates an element $s \in S$. Because the image of the operation is contained entirely within S , an operation satisfying this definition is said to be *closed*. For example, $+$ is a closed operation on \mathbb{Z} , but \div is not; the sum of any two integers is an integer, but $\frac{1}{2}$ is not an integer. All operations considered here are closed on the underlying set for which they are defined.

With the exception of a few explanatory examples, all the algebraic structures we consider here will be defined on a *finite* set S , i.e., a set with only finitely many elements. Although non-finite fields are a rich and rewarding area of mathematical study in their own right, they are not used for any of the cryptographic protocols we will discuss.

2.1.1 Groups

Definition 2.1. A *group* is a pair (G, \cdot) , where G is a set and \cdot is a binary operation on G such that

1. \cdot is associative; that is, $\forall a, b, c \in G$,

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

2. There is an identity element $e \in G$ such that $\forall a \in G$,

$$a \cdot e = e \cdot a = a$$

3. For each $a \in G$, there exists an inverse element $b \in G$ such that

$$a \cdot b = b \cdot a = e$$

This element b is usually denoted a^{-1} .

Furthermore, it can be proved that e is unique, and that for each $a \in G$, the corresponding inverse element a^{-1} is unique. For details, see Dummit and Foote [16, section 1.1, Proposition 1].

For convenience, the same symbol used for the underlying set is often used to denote the group; it is common to see a group (G, \cdot) referred to as “the group G ” when the operation has already been specified or is clear from context.

If the group operation is commutative, that is, if $\forall a, b \in G$ the property $a \cdot b = b \cdot a$ holds, then the group is called *abelian* or commutative.

The dot is often omitted, writing simply ab for $a \cdot b$, and repeated composition of an element $a \in G$ with itself, that is, $aa \cdots a$ with $n \in \mathbb{N}$ copies of a , is written as a^n . The identity element e is commonly named 1. This is intuitive when the group operation is multiplicative in nature, for example in (\mathbb{Z}, \times) , but it can seem strange when the group operation is additive, for example in $(\mathbb{Z}, +)$. In those cases, there is alternative notation available: $+$ for the group operation, na for n -fold composition of $a \in G$ with itself, $-a$ rather than a^{-1} for the inverse of a , and 0 for the identity element. When this additive notation is used, juxtaposition of elements to indicate application of the group operation is avoided; $a + b$ is *not* written as ab .

Table 2.1. Multiplicative and additive notation for groups

	Multiplicative notation	Additive notation
Group operation	$a \cdot b$ or ab	$a + b$
n -fold composition	a^n	na
Inverse	a^{-1}	$-a$
Identity element	1	0

This is ultimately mere notational convention, but since the choice of notation is usually based on whether the group operation behaves more like addition or more like multiplication according to a mathematician’s prior experience with algebraic structures, the choice can aid (or hinder) comprehension. Additionally, having both types of notation available is useful when working with rings and fields, which have both an additive and a multiplicative binary operation defined on their underlying set. Additive notation tends to only be used for abelian groups, since non-commutative additive operations are relatively rare whereas

non-commutative multiplicative operations are very common—the canonical example is multiplication of square matrices, discussed below.

Many familiar algebraic structures are groups. For example:

1. $(\mathbb{Z}, +)$, the integers equipped with the customary addition operation, is an abelian group [29, Example 1.2]. Addition is associative and commutative, the identity element is 0, and the inverse of any element $n \in \mathbb{Z}$ is $-n \in \mathbb{Z}$.
2. $(\mathbb{Q} \setminus \{0\}, \cdot)$, the nonzero rational numbers equipped with the customary multiplication operation, is an abelian group [16, Section 1.1]. Multiplication is associative and commutative, the identity element is 1, and whenever $a, b \in \mathbb{Q} \setminus \{0\}$, the element $\frac{a}{b} \in \mathbb{Q}$ has an inverse $\frac{b}{a} \in \mathbb{Q}$.
3. The set of invertible square matrices over \mathbb{R} , equipped with the usual matrix multiplication operation, gives an example of a non-abelian group [16, Section 1.4]. For an n -by- n matrix it is called the general linear group of degree n and denoted $GL_n(\mathbb{R})$ or in some texts $GL(n, \mathbb{R})$. For example, in $GL_2(\mathbb{R})$, the multiplicative group of invertible 2-by-2 matrices, $\begin{pmatrix} 0 & 2 \\ 4 & 6 \end{pmatrix} \begin{pmatrix} 4 & 3 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 4 & 2 \\ 28 & 18 \end{pmatrix}$, but $\begin{pmatrix} 4 & 3 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} 0 & 2 \\ 4 & 6 \end{pmatrix} = \begin{pmatrix} 12 & 26 \\ 4 & 10 \end{pmatrix}$.
4. $(\mathbb{Z}_n, +)$, the integers modulo n equipped with modular addition [29, Definition 1.5]. This group is very often referred to just as \mathbb{Z}_n . (Note that \mathbb{Z}_n is also written as $\mathbb{Z}/n\mathbb{Z}$ in some texts.)

The familiar case of “clock arithmetic” serves as an introduction to the more general concept of modular arithmetic. When the number of minutes reaches 60, it “wraps back around” to 0 instead of continuing to count up, and similarly the number of hours wraps around at 24 (or 12, depending on the system). 65 minutes after 6 o’clock is 7:05, not 6:65—the part after the colon is the part we care about—and 20 minutes before 8:10 is 7:50, not 8:-10. In other words, when counting minutes on a clock, $0 + 65 = 5$ and $10 - 20 = 50$. This group is called \mathbb{Z}_{60} . In general, $\mathbb{Z}_n = \{0, 1, 2, \dots, n - 1\}$, and for any $x \in \mathbb{Z}$, $x \bmod n$ is the remainder when x is divided by n . Using the examples just given, we would write $65 = 5 \bmod 60$ and $10 - 20 = 50 \bmod 60$.

It is straightforward to see that this operation is associative and commutative, but the existence of inverses and an identity element requires some special consideration. An identity element for addition mod n is 0, but it is also n , or $2n$, or $-5n$, or any other integer multiple of n . This seems to present a problem for considering this structure as a group: after all, the identity element of a group must be unique. A similar problem arises with inverse elements: if $5 + 2 = 0 \bmod 7$, but $5 + 16 = 0 \bmod 7$ as well, how are we to know what “the” additive inverse of 5 is when working modulo 7?

The solution to this problem is that formally, the underlying set \mathbb{Z}_n is actually a set

of equivalence classes of integers, not a set of integers. For a more thorough treatment of modular arithmetic by way of equivalence classes, see [29, Definition 1.4]; for our purposes, it suffices to mention that the elements of \mathbb{Z}_n are actually sets, each of which contains exactly one element of $\{0, 1, \dots, n-1\}$ *as well as* all other integers congruent to it modulo n . The identity element of \mathbb{Z}_{60} is not just 0, but rather $\{\dots, -120, -60, 0, 60, 120, 180, \dots\}$, usually denoted $[0]$. When we work in this group, we can use the canonical representatives and reduce modulo n when necessary: $a + b = [a] + [b] = [a + b] \bmod n$.

A case of particular relevance to us is $(\mathbb{Z}_2, +)$. In this group, $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1$, and $1 + 1 = 0$. This is equivalent to the bit operation XOR, which makes computations in this group exceptionally well-suited to being performed in computer hardware. We will meet \mathbb{Z}_2 again shortly.

After becoming familiar with the additive group $(\mathbb{Z}_n, +)$, it is reasonable to ask whether \mathbb{Z}_n also admits the formation of some kind of multiplicative group. We might begin by testing small values of n to see how they behave under the operation of “first multiply, then reduce modulo n ”. It quickly becomes apparent that 0 poses a problem, since there is no integer k for which $0k = 1 \bmod n$. But what if we exclude 0, similarly to how we exclude *non-invertible 2-by-2 matrices* over \mathbb{R} to obtain the multiplicative group $GL_2(\mathbb{R})$?

A quick check of \mathbb{Z}_2 and \mathbb{Z}_3 appears promising. In the former, the only nonzero element is 1, and it is straightforward to see that $1 \cdot 1 = 1 \bmod 2$ so $1^{-1} = 1 \bmod 2$. In the latter, $1^{-1} = 1$ again, and since $2 \cdot 2 = 4 = 1 \bmod 3$, we have $2^{-1} = 2 \bmod 3$. However, \mathbb{Z}_4 stymies our efforts: $2 \cdot 1 = 2 \bmod 4$, $2 \cdot 2 = 4 = 0 \bmod 4$, and $2 \cdot 3 = 6 = 2 \bmod 4$. There are no more elements to check, so we can conclude that 2 has no multiplicative inverse modulo 4.

As it turns out, the elements in \mathbb{Z}_n which have multiplicative inverses are precisely those which are relatively prime to n [36, Theorem 8.2.1].

Definition 2.2. The multiplicative group of integers modulo n is denoted \mathbb{Z}_n^* or sometimes $(\mathbb{Z}/n\mathbb{Z})^\times$, and its underlying set is $\{a \in \mathbb{Z}_n \mid \gcd(a, n) = 1\}$.

For example, $\mathbb{Z}_5^* = \{1, 2, 3, 4\}$, but $\mathbb{Z}_4^* = \{1, 3\}$; as we just saw, 2 has no multiplicative inverse when working modulo 4. Essentially, if n is prime then $\mathbb{Z}_n^* = \{1, 2, \dots, n-1\}$, but if n is composite then some elements will be “missing”. Just like with \mathbb{Z}_n , the elements of \mathbb{Z}_n^* are technically congruence classes of integers, not “plain” integers, but it is customary to be rather cavalier about it and refer to the integer a and the congruence class $[a]$ interchangeably.

\mathbb{Z}_n^* is at the heart of the RSA cryptosystem, so we will explore it in more detail when we describe RSA in subsection 2.3.2.

Subgroups and cyclic groups

Here we only present a few definitions necessary to understand some material that will be developed later, and brief examples. A more detailed treatment of subgroups, cyclic groups, and generators can be found in sections 2.3 and 2.4 of Dummit and Foote [16] or in sections 2.2 through 2.4 and section 7.10 of Artin [2].

Definition 2.3. A subset H of a group G is called a *subgroup* if it is also a group under the same operation. Since associativity under the group operation follows from $H \subseteq G$, what must be checked in practice is that H is closed: if $a, b \in H$ then $ab \in H$, and if $a \in H$ then $a^{-1} \in H$.

For example, $\{0, 2, 4\}$ is a subgroup of \mathbb{Z}_6 .

Definition 2.4. For a group G and an element $x \in G$, the *subgroup generated by x* is the set of all powers of x , and it is in fact a subgroup of G . It is written $\langle x \rangle$.

Definition 2.5. If x generates the entire group G —that is, $\langle x \rangle = G$ —then x is called a *generator* or *primitive element* of G .

For example, in \mathbb{Z}_9 , $\langle 2 \rangle = \{0, 2, 4, 6, 8, 1, 3, 5, 7\} = \mathbb{Z}_9$ and $\langle 3 \rangle = \{0, 3, 6\}$, so 2 is a primitive element in this group but 3 is not.

Definition 2.6. A group which can be generated by a single element is called *cyclic*.

For example, $\mathbb{Z}_{14}^* = \{1, 3, 5, 9, 11\}$ can be generated by 3, so it is cyclic, but $\mathbb{Z}_8^* = \{1, 3, 5, 7\}$ cannot be generated by any single individual element (try it yourself!), so it is not a cyclic group.

Cyclic groups will become important in section 2.2.

2.1.2 Rings

Definition 2.7. A *ring* is a triple $(R, +, \cdot)$ where R is a set and $+$ and \cdot are two binary operations defined on R such that:

1. R is an abelian (commutative) group with respect to $+$.
2. The operation \cdot is associative.
3. The *distributive laws* hold: that is, $\forall a, b, c \in R$ we have $a(b + c) = ab + ac$ and $(a + b)c = ac + bc$.
4. R has a multiplicative identity element: that is, $\exists e \in R$ such that $\forall a \in R$, $ea = ae = a$.

By convention, the identity element for R under $+$ is denoted 0, and the identity element for R under \cdot is denoted 1.

Similarly to groups, the ring $(R, +, \cdot)$ is often referred to as “the ring R ” if there is no confusion about which operations are defined on it.

Note that although $+$ must be commutative, \cdot is not required to be commutative. However, it may be, and if it is, then the ring $(R, +, \cdot)$ is referred to as a *commutative ring*.

Some authors, including Lidl and Niederreiter [29, Definition 1.28], omit the requirement for R to have a multiplicative identity element, and refer to rings that do contain one as “rings with identity”. Authors who do require rings to have a multiplicative identity element (as is the convention taken in this thesis) sometimes refer to rings without one as “rngs”. We will not discuss rngs further, since they are not relevant to any of the cryptographic techniques presented here. Rings are covered in detail in Part II of Dummit and Foote [16] and Chapter 11 of Artin [2].

$(\mathbb{Z}_n, +, \cdot)$ is a particularly important ring in cryptography: it is the algebraic structure used in RSA, which we will demonstrate in subsection 2.3.2.

2.1.3 Fields

Definition 2.8. A *field* is a ring $(R, +, \cdot)$ which has the following additional properties:

1. The operation \cdot is commutative.
2. $(R \setminus \{0\}, \cdot)$ forms a group.

In other words, with the exception of the additive identity 0 , every element $a \in R$ has a multiplicative inverse $a^{-1} \in R$.

A common illustrative example of a field is $(\mathbb{Q}, +, \cdot)$, the rational numbers equipped with the usual operations of addition and multiplication [18, Section 11.4]. However, due to the difficulty of performing fast computer arithmetic on rational numbers, this field is of limited cryptographic interest. For cryptographic purposes, integers are much easier to work with. We saw in subsection 2.1.1 that for p prime, \mathbb{Z}_p is a group under modular addition and $\mathbb{Z}_p \setminus \{0\}$ is a group under modular multiplication; the commutativity of modular multiplication follows from the commutativity of integer multiplication in general, so \mathbb{Z}_p forms a field in a very natural way. The simplest integer field of all is \mathbb{Z}_2 , where the underlying set is $\{0, 1\}$, addition is binary XOR, subtraction is the same operation as addition, and multiplication is binary AND.

In addition, there are more finite fields than just the ones of prime order. We also have the following result, which will enable us to use the computer-friendly properties of \mathbb{Z}_2 to operate on a much larger set than just $\{0, 1\}$.

Theorem 2.1. *For every prime p and every positive integer m , there exists a finite field with $q = p^m$ elements. Furthermore, for each q , all fields of order q are isomorphic. [29,*

Theorem 2.5]

The isomorphism guaranteed by the second part of Theorem 2.1 means that for a given size q , we are justified in referring to “the” field of order q . The choice of which representation to use for a given field in cryptosystem implementation is a practical one: some are easier to perform computations with than others. The fields are named for the mathematician Évariste Galois, whose work laid the foundations for their study; an account of Galois’ work in its historical context can be found in Edwards [17].

Definition 2.9. The *Galois field* of order q is the finite field of order q . It is commonly denoted $GF(q)$ or \mathbb{F}_q .

We saw earlier that $\mathbb{Z}_4 \setminus \{0\}$ does not form a multiplicative group. However, $4 = 2^2$, a prime power, so Theorem 2.1 tells us that there must be *some* way to construct a field of 4 elements, or 8, or 16, or 256, and so on. What kind of elements and what kind of operation should we use, since modular integer arithmetic isn’t up to the task for $m \geq 2$?

Finite fields of non-prime order

A convenient way to represent and operate on the elements of the finite field of order p^m for p prime and $m \geq 2$ is as polynomials of maximum degree $m - 1$ with coefficients in $GF(p)$. Instead of modular reduction being performed by taking the remainder of an integer after division by p , it is performed by taking the remainder of a polynomial after division by some *irreducible polynomial* $P(x)$, a polynomial which has as its only factors 1 and itself [29, Chapter 2]. For example, $x^2 + x + 1$ is irreducible, but $x^2 + 1$ is not, since over $GF(2)$ it is equivalent to $(x + 1)^2$ (the middle term $2x$ of the product vanishes because $2 = 0 \pmod{2}$).

The elements of $GF(2^m)$ can be represented by bit strings of length m . For example, the string 1011 corresponds to the polynomial $x^3 + x + 1$ in $GF(16)$.

Irreducible polynomials for $GF(2^m)$ are chosen to be of the form $x^m + x^k + 1$ when possible, preferring the lowest possible value of k . For some field sizes, all polynomials of this form are reducible; in those cases it is necessary to choose a polynomial with 5 terms, rather than 3, but more than 5 are never needed [12, p. 10].

Addition and subtraction are the same operation on polynomials over $GF(2^m)$, so without loss of generality we will only consider addition. It is performed componentwise, just like the “usual” polynomial addition taught to children, and in the bit string representation it corresponds to XOR:

$$\begin{array}{rcccc}
 & 1 & 0 & 1 & 1 & & x^3 & & +x & +1 \\
 \oplus & 1 & 1 & 0 & 1 & & + & x^3 & +x^2 & +1 \\
 \hline
 & 0 & 1 & 1 & 0 & & & & x^2 & +x
 \end{array}$$

Note how this differs from modular addition of integers: if these were elements of \mathbb{Z}_{16} , the result would be $1011_2 + 1101_2 = 11 + 13 \equiv 8 = 1000_2 \pmod{16}$.

Multiplication is performed in the usual manner and then followed by modular reduction by the irreducible polynomial $P(x)$. In other words, the product of two polynomials $A(x)$ and $B(x)$ is the polynomial $C(x)$ such that $A(x) \cdot B(x) \equiv D(x) \cdot P(x) + C(x)$, where the degree of $C(x)$ is strictly less than the degree of $D(x)$. This remainder $C(x)$ always exists, and can be computed using polynomial long division. [29, p. 20]

As an example, let's compute the product of $A(x) = x^3 + x^2 + 1$ and $B(x) = x^2 + x + 1$ in $GF(16)$ with the irreducible polynomial $P(x) = x^4 + x + 1$. The product $A(x) \cdot B(x)$ is $x^5 + x + 1$ (remember that the coefficients are elements of $GF(2)$, so addition is performed mod 2). To compute the remainder $C(x)$, we can perform long division:

$$\begin{array}{r}
 x^4 + x + 1 \overline{) x^5 + x + 1} \\
 \underline{x^5 + x} \\
 x^2 + 1
 \end{array}$$

The remainder—i.e., the result of the multiplication in $GF(2^4)$ —is $x^2 + 1$.

Although not applicable to the present work, it is mathematically interesting to note that there are at least two other ways to represent the elements of these fields: one using cyclotomic polynomials, and one using matrices. These are covered in [29, Chapter 2, Section 5].

2.2 Elliptic curves

A non-supersingular¹ elliptic curve E over a binary finite field $GF(2^m)$ is the set of points (x, y) , where $x, y \in GF(2^m)$, which satisfy an equation of the form

$$y^2 + xy = x^3 + ax^2 + b$$

¹Supersingularity is a property of some elliptic curves which makes them undesirable for most cryptographic applications because their *discrete logarithm problem* (see subsection 2.3.3), on which their security depends, is easier to solve. [40, p. 388]

for some particular constants $a, b \in GF(2^m)$. $E(GF(2^m))$ or $E/GF(2^m)$ are the usual ways of writing “the curve E over the field $GF(2^m)$ ” concisely. Hankerson, Vanstone, and Menezes [22] is an excellent guide to elliptic curves with a specific eye towards their use in cryptography, including a discussion of implementation considerations; Silverman and Tate [41] present the underlying mathematics in detail, and treat cryptographic applications briefly in Chapter 4.

There is a customary way to form an additive cyclic group using the points (x, y) on E taken together with a “point at infinity” which is written \mathcal{O} or ∞ . The tradition is to use additive notation and terminology for this group, probably because it is abelian, but since the operation bears little resemblance to familiar notions of addition or multiplication, this choice is more or less arbitrary. Note that this is very different from the case of $(\mathbb{Z}_n, +)$, where the choice of additive notation is motivated by the close relationship between the group operation and the “usual” addition operation over \mathbb{Z} .

To develop a “feel” for the group operation, it is easiest to give a geometric description for how it works for an elliptic curve over \mathbb{R} , and then note that the geometric operations can also be described algebraically in terms of the coordinates of the points involved and that these algebraic formulae work just as well for curves over finite fields as they do for curves over \mathbb{R} even though the geometric intuition no longer applies.²

Elliptic curves over fields other than $GF(2^m)$ or $GF(3^m)$ are usually given by an equation of the slightly simpler form $y^2 = x^3 + ax + b$ for some constants a, b in the underlying field [22, Section 3.1.1]. For example, Figure 2.1 illustrates the curve over \mathbb{R} or \mathbb{Q} given by the equation $y^2 = x^3 - 3x + 5$, and an example of the group operation on it.

- The sum $R = P + Q$ of two points P and Q is computed by finding the third point of intersection of the line through P and Q with the curve E , and then reflecting that point over the x axis. If $P = Q$, then $R = P + P = 2P$ is the reflection of the second point of intersection of the tangent line through P with the curve E ; this special case is called “point doubling”.
- \mathcal{O} is the identity element: $\mathcal{O} + P = P + \mathcal{O} = P$ for every point P .
- $-P$, the inverse of P , is obtained by reflecting P over the x axis. In particular, $-\mathcal{O} = \mathcal{O}$.

²For a more in-depth but still informal and approachable description of the geometric approach, see [41, Chapter 1] (although be aware that the initial discussion in section 1.2 uses \mathcal{O} to denote a point *on* the curve rather than an additional point at infinity); for a terse but mathematically precise definition of how and why the curve equation may differ in form and how the algebraic formulae for the group operation are derived depending on the underlying field, see [22, Chapter 3].

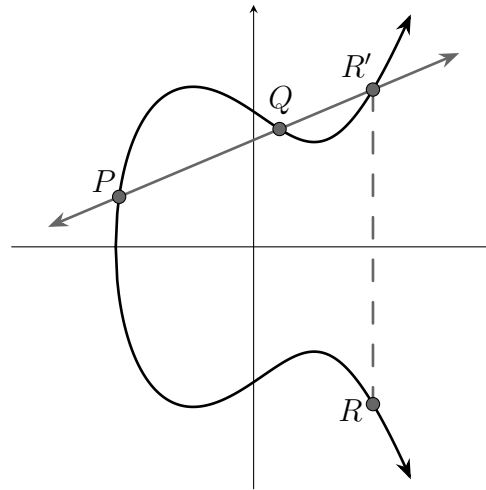


Figure 2.1. Point addition on the elliptic curve $y^2 = x^3 - 3x + 5$ over \mathbb{R} or \mathbb{Q}

The geometric intuition which links the inverse and the identity element should hopefully be clear: when P and Q have the same x coordinate, the line through them is vertical, so its “third point of intersection with the curve” is the point at infinity, and therefore $P + Q = \mathcal{O}$, which means Q must be $-P$.

In keeping with the use of additive notation, exponentiation in this group is written nP rather than P^n , and called “point multiplication” (sometimes “scalar point multiplication” to emphasize that we are “multiplying” a point by an integer rather than by another point, although this clarification is technically unnecessary since there is no multiplication operation of one point by another point from which it would need to be distinguished). Unlike in the case of an additive group over the integers, we cannot use any “extra” mathematical structure beyond the group operation to perform this operation quickly; however, we *can* use fast exponentiation techniques such as the square-and-multiply/double-and-add algorithm, in which we begin with the point P and then scan the binary digits of the exponent (excepting the leading one) from left to right, at each step first doubling the value (no matter what), then adding one more copy of P only if the digit is a 1. For example, to compute $29P$ ($29_{10} = 11101_2$), we would double-and-add twice, then double, then double-and-add again: $P \rightarrow 3P \rightarrow 7P \rightarrow 14P \rightarrow 29P$. This is much faster than performing 28 separate additions.

2.3 Cryptographic applications

These mathematical structures may be very interesting, but in order to be cryptographically relevant, they need more than just to be enjoyable to study in their own right: they need to enable secure private communication across an unsecured channel. Prior to the 1970s, all cryptosystems were symmetric cryptosystems: they relied on both parties physically exchanging a shared secret key which they would then use to encrypt and decrypt messages. Today, asymmetric cryptosystems based on the previously described

algebraic structures make it possible for two parties to create a shared piece of secret information without ever having to meet in person. However, these asymmetric schemes are computationally expensive compared to symmetric schemes, so they would be prohibitively slow to use for e.g. encrypting large files or streaming video. For that reason, the typical modern cryptographic exchange involves first using an asymmetric algorithm to agree on a shared secret key, and then using that key in a symmetric algorithm to encrypt and decrypt the actual contents of the messages being exchanged [36, chapter 6].

Asymmetric schemes can also provide *nonrepudiation*, the capability to cryptographically sign a message in a way that enables anyone to verify that it was really sent by the holder of a certain public key. Of course, the question of verifying that a certain public key really belongs to a certain person or other entity in the real world is in part a social question of which authorities to trust, which cannot be solved with mathematics alone; for a discussion of the complexities involved, see Paar and Pelzl [36, section 13.3].

2.3.1 One-way functions

The cornerstone of public-key cryptography is the concept of a *one-way function*.

Definition 2.10. A function $f : X \rightarrow Y$ is a one-way function when:

1. f is invertible: each $y \in Y$ is associated to a *unique* input $x \in X$
2. $y = f(x)$ is relatively easy to compute
3. $x = f^{-1}(y)$ is prohibitively difficult to compute

The third point comes with one caveat: it only holds when the sets X and Y are sufficiently large. If there are too few possible input-output pairs, then any easy-to-compute function can be inverted by brute force search, even if there are no known efficient algorithms for computing the inverse in general. This is why the numbers (and other types of elements in groups, rings, and fields) used in modern cryptography are hundreds or even thousands of bits long.

It tends to be the case that there is *not* a mathematical proof of the hardness of computing f^{-1} . This is the situation for both of the one-way functions we are about to discuss (integer factorization and the discrete logarithm problem), which underlie the most widely used cryptosystems in the world today [22, Chapter 1]. This may be surprising, since it means that we all depend every day on systems that might in theory be vulnerable to attacks that just haven't been discovered yet—not only in the specifics of how they are implemented and deployed (which of course is always a risk), but in the underlying mathematics itself! The widespread agreement that there is no easy method for inverting these functions is extremely unlikely to exist comes not from formal proofs, but instead from the fact that the combined efforts of many large and highly skilled research communities at

the cutting edge of modern mathematics have so far failed to discover any. It remains, in theory, possible that one might be found—and if it were, it would have far-reaching consequences for many fields of mathematics and computer science, not only cryptography.

Just like with the question mentioned above of how to verify that a public key really belongs to a certain person, this is an area where we must rely not only on the mathematics itself, but on ultimately subjective judgments about the trustworthiness of other people, communities, and authorities.

2.3.2 The integer factorization problem and RSA

RSA is an asymmetric cryptosystem which relies on the fact that it is computationally easy to multiply large integers but (apparently) computationally very difficult in general to factor them. The mathematical setting for RSA is the integer ring \mathbb{Z}_n with the usual operations of modular addition and modular multiplication, where n is the product of two large primes p and q . RSA is one of the most widely used asymmetric cryptographic schemes, as well as one of the oldest, and it is named for the three people who first publicly described it: Ron Rivest, Adi Shamir, and Leonard Adleman [36, p. 173].

Before describing RSA, it is necessary to introduce one more definition, related to the ring $(\mathbb{Z}_n, +, \cdot)$.

Definition 2.11. *Euler's totient function*, also known as *Euler's phi function*, is the number of elements in \mathbb{Z}_n which are relatively prime to n . It is denoted $\varphi(n)$ or sometimes $\Phi(n)$.

It is always possible in theory to compute $\varphi(n)$ by actually computing $\gcd(a, n)$ for each $a \in \mathbb{Z}_n$ and counting how many of them are equal to 1. This is far too slow to be useful in practice at cryptographic sizes, even with large amounts of computing power. However, *if* we know the prime factorization of n , then we can compute $\varphi(n)$ very quickly with the following formula: when $n = \prod_{i=1}^k p_i^{n_i}$ is the canonical prime factorization of n , the totient of n is given by the formula $\varphi(n) = \prod_{i=1}^k (p_i^{n_i} - p_i^{n_i-1})$.

In particular, for a prime p , $\varphi(p) = p^1 - p^0 = p - 1$. (This should make intuitive sense: there are p elements in \mathbb{Z}_p , and only one of them—0—is not relatively prime to p , so $p - 1$ elements of \mathbb{Z}_p are relatively prime to p .) For p, q prime and $n = pq$, as we will have in RSA, $\varphi(n) = (p - 1)(q - 1)$.

The security of RSA is conditional not only on it being the case that integer factorization is hard, but also on it being the case that there is no significantly faster way to compute $\varphi(n)$ than by factoring n .

An RSA keypair consists of a public key (n, e) and a private key d . This is how the keypair is generated:

Setup Choose two large primes p and q .

Public key computation Compute $n = pq$. Compute $\varphi(n) = (p - 1)(q - 1)$. Select a public key exponent $e \in \{1, 2, \dots, \varphi(n) - 1\}$ which is relatively prime to $\varphi(n)$ —that is, $\gcd(e, \varphi(n)) = 1$.

Private key computation Compute the private key d such that $de = 1 \pmod{\varphi(n)}$.

The private key $d = e^{-1} \pmod{\varphi(n)}$ can be computed very efficiently with the Extended Euclidean Algorithm [36, section 6.3.2]. This is why RSA depends on the computational intractability of determining $\varphi(n)$ without knowing the factorization of n : if someone discovered an efficient way to compute $\varphi(n)$, then they would be able to obtain the corresponding private key for any public key!

Encryption and decryption in RSA is straightforward:

Encryption To encrypt $x \in \mathbb{Z}_n$, compute $y = x^e \pmod{n}$.

Decryption To decrypt $y \in \mathbb{Z}_n$, compute $x = y^d \pmod{n}$.

It should be apparent that in order for this scheme to work, it must be the case that $x = (x^e)^d \pmod{n}$. This equivalence follows from a property known as Euler's Theorem: if a and m are integers with $\gcd(a, m) = 1$, then $a^{\varphi(m)} = 1 \pmod{m}$. For details, see Paar and Pelzl [36, section 7.3]. (Euler's Theorem itself is not proved there, but the interested reader can note that it is a natural consequence of Lagrange's Theorem—a more general result about the possible orders of subgroups of a finite group—applied to the particular case of the multiplicative group \mathbb{Z}_n^* .)

When encrypting or decrypting, it is not necessary to first compute the “full” result x^d or y^e in \mathbb{Z} —which would be extremely large—and wait until the end to reduce it modulo n . Instead, we can perform modular reductions as we go, and thanks to the nice algebraic properties of equivalence classes which underlie modular arithmetic, we never have to work with numbers much larger than n itself. Additionally, it is possible in general to compute x^d much faster than the naive method of multiplying x with itself d times (and similarly for y^e), by taking advantage of the square-and-multiply algorithm for fast exponentiation. [36, section 7.4] [19, section 4.3]

2.3.3 The discrete logarithm problem and Diffie–Hellman–Merkle

Besides modular integer multiplication, another function which is strongly believed to be one-way and therefore sees widespread use in cryptosystem design is exponentiation in a finite cyclic group. For a finite cyclic group G of order n , where a is an arbitrary element in G and g is a primitive element in G , the *discrete logarithm problem* is the problem of finding the integer $x \in \{1, 2, \dots, n\}$ such that $a = g^x$. By analogy with the case of exponentiation in \mathbb{R} , we denote $x = \log_g a$. Diffie–Hellman–Merkle Key Exchange

(DHMKE)³ is a cryptographic scheme that takes advantage of the property $(g^x)^y = (g^y)^x$, and the difficulty of computing x from g and g^x , to establish a shared secret between parties communicating over an insecure channel. The original publications are [33] and [15].

Over \mathbb{R} , logarithms can be computed quickly (or at least approximated as closely as desired), since all that needs to be done is to invert the exponential function, which is continuous and monotonic: for some fixed positive number $a \neq 1$, a^x and a^y are close exactly when x and y are also close. Equivalently, $x = \log_a b$ and $y = \log_a c$ are close exactly when b and c are close. However, the behavior in finite cyclic groups is much less predictable. For example, $g = 3$ is a primitive element in the multiplicative group \mathbb{Z}_{31} , but consecutive choices for x in 3^x may produce elements that are far apart or close together: $3^{12} = 8$ and $3^{13} = 24$ in that group, but we also have $3^{21} = 15$ and $3^{22} = 14$.

The reason this is useful for cryptography is not merely that the relationship between successive values of x and successive values of g^x varies non-monotonically, but that it varies *in a way that is difficult to predict*. Care must be taken in the choice of a group for a discrete log cryptosystem, since some finite cyclic groups have additional structure beyond the group operation which allows logarithms in them to be computed efficiently: for example, \mathbb{Z}_p (p prime) with addition rather than multiplication is easy to compute discrete logarithms in, because “exponentiation” in that group is merely modular integer multiplication, and we know how to compute modular multiplicative inverses very quickly [36, Section 8.3.2].

The two most common choices of the group G in DHMKE are \mathbb{Z}_p^* (as used in the original publication describing the scheme [15]) or an elliptic curve E over some finite field $GF(q)$. For the latter case, either prime fields $GF(p)$ or extension fields $GF(p^m)$ can be used.

Note that during the key generation and shared secret computation phases, the private keys are *integers*, the public keys are *elements of the group*, and the exponentiations are of course performed in the group G , so for example in \mathbb{Z}_p^* the generator g is an integer and the computation g^a is $g^a \bmod p$, and in an elliptic curve group E the generator g is a point on the curve and the computation g^a is the scalar point multiplication ag .

Setup A finite cyclic group G and a generator $g \in G$ are chosen.

Key generation Alice chooses a private key $a = k_{pr,A} \in \{2, 3, \dots, n-1\}$ (where n is the order of G), and computes her public key $A = k_{pub,A} = g^a$. Bob chooses a private key $b = k_{pr,B} \in \{2, 3, \dots, n-1\}$, and computes his public key $B = k_{pub,B} = g^b$. Alice publishes A and Bob publishes B .

Shared secret computation Alice uses Bob’s public key B to compute $k_{BA} =$

³The scheme first became known by the name “Diffie–Hellman key exchange”, but Hellman has stated [23] that Merkle’s name ought to be included as well, in recognition of Merkle’s foundational contributions to the early development of public-key cryptography.

B^a . Bob uses Alice's public key A to compute $k_{AB} = A^b$. Because $B^a = (g^b)^a = (g^a)^b = A^b$, $k_{BA} = k_{AB}$, and therefore they have computed a shared secret.⁴

Note that although the security of DHMKE is considered to depend on the hardness of the discrete logarithm problem, it could be broken without solving the discrete logarithm problem in general if an efficient way to compute g^{ab} from the triple (g, g^a, g^b) were discovered. This is not exactly the same problem as computing x from the pair (g, g^x) . This is similar to the case with RSA, which could theoretically be broken without solving the integer factorization problem if someone were to discover a way to compute $\varphi(n)$ quickly without factoring n . Even setting this somewhat curious property aside, RSA and DHMKE are remarkably similar in structure, and in fact, this is no accident: integer factorization and the discrete logarithm problem are both instances of the *hidden subgroup problem*, a more general group-theoretic problem [35, section 5.4.3].

Despite the mathematical similarities, there is a major practical difference in how these two hidden subgroup cryptosystems are used in practice: unlike in RSA, where fresh primes p and q are chosen every time a new keypair is generated, the domain parameters used in DHMKE—a large prime p and an element of \mathbb{Z}_p^* in the integer case, or a curve $E(GF(q))$ with certain cryptographically desirable properties and a point on it in the elliptic curve case—are rather expensive to compute. Therefore, the process of generating a shared secret via DHMKE usually begins not with choosing domain parameters “on the fly”, but rather with the selection of a predefined group which has been published by some external authority. Common parameters for integer field DHMKE are available from the Internet Engineering Task Force (IETF) [26, 20], and common named curves for elliptic curve DHMKE are published by the National Institute of Standards and Technology (NIST) [12].

⁴Paar and Pelzl [36] call both of these values k_{AB} from the start, but I consider it more instructive to give them different names at first, and only refer to them by the same name after showing why they are equal.

3. POLYNOMIAL MULTIPLICATION

Efficient multiplication is of great importance to the practical implementation of finite field cryptosystems. This chapter considers how the state of the art in fast multiplication has evolved since the first real breakthrough on the problem in the 1960s, with particular attention paid to the algorithms used in `gf2sliced`. Other algorithms are briefly discussed in the final section.

Most of these algorithms are designed and presented for working on large integers (“large” here generally meaning “too big to fit in whatever integer data type the machine uses natively”, e.g. larger than 64-bit on many modern machines), rather than on polynomials. However, the underlying operations are generalizable from integers to polynomials in consistent ways. This generalization process is discussed in section 3.3.

3.1 Schoolbook multiplication

In school, most students learn the following simple algorithm for multiplying two polynomials $p(x)$ and $q(x)$: multiply each term of $p(x)$ by each term of $q(x)$ and add all the resulting terms together. Formally, given two polynomials $p(x) = \sum_{i=0}^m p_i x^i$ and $q(x) = \sum_{j=0}^n q_j x^j$, their product is $p(x)q(x) = \sum_{i=0}^m p_i x^i (\sum_{j=0}^n q_j x^j) = \sum_{i=0}^m \sum_{j=0}^n p_i q_j x^{i+j}$.

When working by hand, the computation often proceeds from the highest-order terms down to the lowest-order term, in accordance with the order in which the terms are written from left to right. For example, the multiplication of $p(x) = 2x^2 + 3x + 5$ by $q(x) = 7x + 1$ proceeds as follows:

$$\begin{aligned} (2x^2 + 3x + 5)(7x + 1) &= 2x^2(7x + 1) + 3x(7x + 1) + 5(7x + 1) \\ &= 14x^3 + 2x^2 + 21x^2 + 3x + 35x + 5 \\ &= 14x^3 + 23x^2 + 38x + 5 \end{aligned}$$

It is illustrative to compare this to the computation of $235 \cdot 71$, where the coefficients of p and q are taken as decimal digits (note that $235 = p(10)$ and $71 = q(10)$). When 7 was considered as a coefficient of x and multiplied by 5, we kept the result $35x$ together “in one piece”, but when considering 7 as an integer in the tens place and multiplying it by 5, we must write down only the 5 in the tens place and carry the 3.

Importantly, note that if we plug $x = 10$ into the resulting polynomial $p \cdot q$, we find that $14 \cdot 10^3 + 23 \cdot 10^2 + 38 \cdot 10 + 5 = 16685$, which is the same result we get when multiplying $235 \cdot 71$ directly as integers. This correspondence is bidirectional, and allows us to use recursive algorithms for integer multiplication easily for polynomials with integer coefficients in the following sections.

This algorithm excels for pencil-and-paper computations for several reasons. It is straightforward to perform by hand; it is in obvious correspondence with the actual definition of the product of two polynomials via the distributive property; and it prescribes the same sequence of steps in all cases, rather than requiring the person or computer using it to perform any kind of analysis on the inputs in advance.

However, it is not very efficient. To multiply two polynomials of n terms each, this algorithm requires n^2 individual multiplications—in other words, its asymptotic complexity is $O(n^2)$ (details on O -notation can be found in section 3.4 for those who are unfamiliar with it). This is fine for polynomials of low degree, but at cryptographically relevant sizes it is rather slow. Can we do better?

3.2 Karatsuba multiplication

If multiplication is computationally expensive and addition is computationally cheap, then it stands to reason that a divide-and-conquer algorithm for multiplication could be asymptotically faster than the one described above, if only it were possible to replace some of those n^2 multiplications with computations that only require addition. For centuries it was widely believed, but never proven, that it was impossible to do better than the schoolbook algorithm. In 1960, famous mathematician Andrey Kolmogorov issued a challenge to the mathematics community to formally prove that no better algorithm could exist—but while working on the problem, Anatoly Karatsuba discovered one! The algorithm he devised, first published in 1962, was a milestone in the study of algorithmic complexity theory [25].

An n -digit number can be split into two halves of length $n/2$ when n is even, or of length $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively when n is odd. For convenience we will consider the case when n is even. In our usual base 10, this means that a number x can be written as $10^{n/2}a + b$, where a is the higher-order $n/2$ digits and b is the lower-order $n/2$ digits. For example, if $x = 5678$, then $n = 4$, $a = 56$, and $b = 78$.

We can compute the product of x and some other number $y = 10^{n/2}c + d$ as follows:

$$\begin{aligned} xy &= (10^{n/2}a + b)(10^{n/2}c + d) \\ &= 10^n ac + 10^{n/2}(ad + bc) + bd \end{aligned}$$

At this point, we could compute the three parts ac , $ad + bc$, and bd separately, and then

combine them in this formula. This would be a recursive algorithm, differing in form from the iterative one in the previous section. But would it be significantly faster?

Unfortunately not, at least not asymptotically. Although it splits the problem into three subproblems, which may seem more efficient than performing four multiplications, the middle subproblem $ad + bc$ requires two multiplications to compute, so the total number of multiplications required remains four. Karatsuba's insight was that the equation can be rewritten as follows:

$$\begin{aligned} xy &= (10^{n/2}a + b)(10^{n/2}c + d) \\ &= 10^n ac + 10^{n/2}(ad + bc) + bd \\ &= 10^n ac + 10^{n/2}(ad + bc + ac + bd - ac - bd) + bd \\ &= 10^n ac + 10^{n/2}((a + b)(c + d) - ac - bd) + bd \end{aligned}$$

Using this formula, we only have to compute three products: ac , bd , and $(a + b)(c + d)$. The additions $a + b$ and $c + d$ are comparatively cheap, and ultimately slip out of consideration when analyzing asymptotic behavior—for details, see Gathen and Gerhard [19, Section 8.1].

For an example, let $x = 5678 = 56 \cdot 10^2 + 78$ and $y = 1234 = 12 \cdot 10^2 + 34$. Then $a = 56$, $b = 78$, $c = 12$, and $d = 34$.

$$\begin{aligned} ac &= 56 \cdot 12 = 672 \\ bd &= 78 \cdot 34 = 2652 \\ (a + b)(c + d) &= (56 + 78)(12 + 34) = 134 \cdot 46 = 6164 \\ (a + b)(c + d) - ac - bd &= 6164 - 672 - 2652 = 2840 \end{aligned}$$

Then $xy = 10^4 \cdot 672 + 10^2 \cdot 2840 + 2652 = 7006654$. The correctness of this result can be verified by hand or calculator.

In the next section, we will see that Karatsuba is in fact a special case of a more general algorithm, where each input may be split into more than two parts.

3.3 Toom–Cook multiplication

“Toom–Cook multiplication” is actually the name of a family of algorithms, which are parameterized by the number of smaller subproblems into which the inputs are split. We call the algorithm “Toom- k ” that splits each input into k subproblems each. (There also exist intermediate algorithms that split the two inputs differently, e.g. Toom-2.5 which splits one of them into 2 pieces and the other into 3 pieces, but we will not discuss them here.) In theory, it would be possible to choose k to be as large as we like (up to the number of digits in the operands—there would be no point in trying to split a 16-bit number into more

than 16 pieces) but in practice this sees diminishing returns, and Toom-4 or occasionally Toom-5 is the most aggressive split generally encountered in the wild. The method was originally described in Steve Cook's PhD thesis; as of this writing, the relevant chapter is available online [13].

The key insight behind the Toom-Cook algorithms is that a polynomial of degree d can be fully specified by only $d+1$ points, and therefore the product of two degree- d polynomials, having degree $2d$, can be specified by only $2d+1$ points. For example, the product of two polynomials of degree 2 (having 3 terms each) will be a polynomial of degree 4, which can be specified by 5 points; since the polynomials we work with represent well-behaved mappings, two polynomials p and q have the property that $(p \cdot q)(a) = p(a) \cdot q(a)$ for any point a in their domain, so $p \cdot q$ can be computed by performing 5 multiplications and then converting the resulting polynomial from point-value form to coefficient form¹, rather than performing the 9 multiplications that would be required to multiply the 3-term-long p by the 3-term-long q in coefficient form directly. There is some overhead incurred in the conversion, but for small values of k it is dominated by the savings obtained from the reduction in the number of multiplications required.

Before describing the inner workings of the algorithm, one matter deserves special attention, owing to its ability to sow confusion if left unaddressed. We aim to multiply polynomials (elements of $GF(2^m)$), and Toom-Cook algorithms use polynomial multiplication to produce their result. However, the mathematical objects that we multiply are *not* the polynomials over $GF(2)$ that we begin and end with (which may have degree up to $m-1$); they are polynomials in a different variable, of degree k corresponding to the choice of algorithm Toom- k , for which the pieces of our input are merely coefficients.

An example will hopefully clarify how this works. Splitting an integer into pieces and using those pieces as the coefficients of a polynomial requires introducing a new variable so that we can keep track of them, and for many mathematicians it feels natural to reach for an x here: we can split 593672 up as $59x^2 + 36x + 72$ without running into any issues at all, as long as we record somewhere that we will need to plug in $x = 10^2$ to get our original number back. But what about when we split up an element of $GF(2^m)$, for example, $x^7 + x^6 + x^3 + x^2 + 1 \in GF(2^8)$? We need to introduce a new variable, and the resulting polynomial will look something like $(x+1)y^2 + xy + x + 1$, and we make a note that $y = x^3 \in GF(2^8)$ is the point at which this polynomial evaluates to the value we started with. This is *not* a multivariate polynomial where x and y have the same domain. This is a *univariate* polynomial in y where the coefficients are elements of $GF(2^8)$ that

¹Coefficient form is the usual way of writing a polynomial by hand, such as $3x^2 + 5$. In the computer, polynomials in coefficient form are generally stored as a list of coefficients, with the lowest-order term first: the polynomial just mentioned would likely be stored as $[5, 0, 3]$. Point-value form means storing a list of $\deg(p) + 1$ distinct coordinates: here we might choose $[(0, 5), (1, 8), (2, 17)]$. Since there are many different ways to represent the same polynomial in point-value form, it is a representation that lends itself well to trying to squeeze more and more performance optimizations out of multiplication algorithms.

just so happen to be written with the letter x , and when we speak of “plugging in” and “evaluating at a point” while working with it, we are always referring to y . It would make no more sense at this point to “evaluate” $x + 1$ at some input value than it would to “evaluate” the number 56 at some input value in our integer example.

The example of the actual operation of Toom-3 will be given with integers, since it is significantly simpler to follow, but it is important to keep this distinction in mind, and any reader who is curious about the details (or unconvinced that this makes any sense at all) is encouraged to work through an example of Toom-3 polynomial multiplication by hand. It can be done with elements of a finite field small enough to be convenient (keep a lookup table on hand for products and inverses!), or with polynomials over \mathbb{Z} . Implementing the algorithm yourself in a computer algebra system such as SageMath or Maple may also be enlightening.

3.3.1 Description of the algorithm

For a detailed and mathematically precise description of Toom–Cook for other values of k and with potentially unbalanced operands (that is, operands which differ significantly in length from each other), see Bodrato [7, Section 2]. Here we present Toom-3 for balanced operands, focusing on explaining the intuition behind it and showing the actual steps of the computation rather than on providing a fully generic formal mathematical specification.

First, split each input into k parts of roughly equal size, and construct the coefficient polynomials p and q , making note of the value at which they will eventually need to be evaluated to recover the initial inputs. The value consists of a base b and an exponent i : in the example above we used 10^2 when splitting an integer into base-10 chunks which contain 2 digits each, and x^3 when splitting a polynomial in x into chunks which contain 3 terms each (including some “terms” with a coefficient of zero). The highest-order part of the input may be shorter if the input length is not a multiple of k , such as when 12345678 is split into $12x^2 + 345x + 678$ at $x = 10^3$. The base and exponent must be the same for both inputs, even if they differ in length.

Second, obtain the product polynomial $r = pq$ in point-value form. For Toom-3, p and q have degree 2, so r will have degree 4, which means we need to compute 5 point-value pairs for p and q . The choice of evaluation points can have a significant impact on performance [7]; for integers, the conventional choice for Toom-3 is $\{0, 1, -1, -2, \infty\}$, and for binary finite fields it is $\{0, 1, x, x + 1, \infty\}$ [8]. “Evaluating at infinity” means taking the limit $\lim_{y \rightarrow \infty} \frac{p(y)}{y^k}$ where k is the degree of p ; this will be the coefficient of the highest-order term of p .

We obtain the following formulae for p in the integer case:

$$\begin{aligned} p(0) &= m_2(0)^2 + m_1(0) + m_0 = m_0 \\ p(1) &= m_2(1)^2 + m_1(1) + m_0 = m_2 + m_1 + m_0 \\ p(-1) &= m_2(-1)^2 + m_1(-1) + m_0 = m_2 - m_1 + m_0 \\ p(-2) &= m_2(-2)^2 + m_1(-2) + m_0 = 4m_2 - 2m_1 + m_0 \\ p(\infty) &= m_2 \end{aligned}$$

The values of q are computed similarly. The question of how to compute these values as efficiently as possible is an interesting one—it is sometimes possible to discover a sequence of arithmetical steps that performs better than simply plugging the value into the polynomial and computing it “the usual way”. Bodrato [7] offers an interesting example of optimizing these calculation steps via graph search. However, on an intuitive level, it is useful to think of this computation as a matrix-vector multiplication:

$$\begin{pmatrix} p(0) \\ p(1) \\ p(-1) \\ p(-2) \\ p(\infty) \end{pmatrix} = \begin{pmatrix} 0^0 & 0^1 & 0^2 \\ 1^0 & 1^1 & 1^2 \\ (-1)^0 & (-1)^1 & (-1)^2 \\ (-2)^0 & (-2)^1 & (-2)^2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} m_0 \\ m_1 \\ m_2 \end{pmatrix}$$

And again similarly for q . The 5-by-3 matrix is sometimes referred to as the evaluation matrix E .

Third, use the computed values of p and q at the 5 evaluation points to obtain a point-value representation of r , by computing $r(0) = p(0)q(0)$ and so on.

Fourth, convert r from point-value form to coefficient form. Here is where the matrix-vector multiplication representation truly shines: by knowing what r evaluates to at each of the 5 evaluation points we chose, we can construct the following matrix equation:

$$\begin{pmatrix} r(0) \\ r(1) \\ r(-1) \\ r(-2) \\ r(\infty) \end{pmatrix} = \begin{pmatrix} 0^0 & 0^1 & 0^2 & 0^3 & 0^4 \\ 1^0 & 1^1 & 1^2 & 1^3 & 1^4 \\ (-1)^0 & (-1)^1 & (-1)^2 & (-1)^3 & (-1)^4 \\ (-2)^0 & (-2)^1 & (-2)^2 & (-2)^3 & (-2)^4 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix}$$

Call this 5-by-5 matrix the interpolation matrix A . Since this choice of evaluation points makes this matrix invertible, it is possible to recover the coefficients of r by multiplying on

the left on both sides by A^{-1} :

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{3} & -1 & \frac{1}{6} & -2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ -\frac{1}{2} & \frac{1}{6} & \frac{1}{2} & -\frac{1}{6} & 2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r(0) \\ r(1) \\ r(-1) \\ r(-2) \\ r(\infty) \end{pmatrix}$$

Although some elements in A^{-1} are fractional, the resulting coefficients of r will always be integers. When working over a finite field, rather than \mathbb{Z} , no such potential concern ever arises: the elements of A^{-1} will all be field elements.

Fifth and finally, evaluate $rb^i = r_4b^{4i} + r_3b^{3i} + r_2b^{2i} + r_1b^i + r_0$.

3.3.2 Example

We will use Toom-3 to compute the product of $m = 654321$ and $n = 234567$.

The first step is to construct the polynomials we will use, and make note of which value we must evaluate them at to recover m and n . It is convenient to choose $p(x) = m_2x^2 + m_1x + m_0 = 65x^2 + 43x + 21$ and $q(x) = n_2x^2 + n_1x + n_0 = 23x^2 + 45x + 67$, using $x = 10^2$ to split our base-10 numbers into 2-digit chunks.

Now we evaluate p and q at the five pre-chosen evaluation points, and compute the five values of r accordingly:

x	$p(x)$	$q(x)$	$r(x) = p(x)q(x)$
0	21	67	1407
1	129	135	17415
-1	43	45	1935
-2	195	69	13455
∞	65	23	1495

The leftmost and rightmost column, taken together, specify the polynomial r in point-value form. Now the coefficients of r can be recovered by considering the rightmost column of outputs as a column vector and multiplying it on the left by the inverse of the matrix A , as discussed in the previous section:

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{3} & -1 & \frac{1}{6} & -2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ -\frac{1}{2} & \frac{1}{6} & \frac{1}{2} & -\frac{1}{6} & 2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1407 \\ 17415 \\ 1935 \\ 13455 \\ 1495 \end{pmatrix} = \begin{pmatrix} 1407 \\ 3826 \\ 6773 \\ 3914 \\ 1495 \end{pmatrix}$$

So $r(x) = 1495x^4 + 3914x^3 + 6773x^2 + 3826x + 1407$. Evaluating it at $x = 10^2$, we obtain the final result 153482114007.

3.3.3 Equivalence of Karatsuba and Toom-2

As mentioned above, Toom-Cook reduces to Karatsuba in the $k = 2$ case. Now that we have seen how both algorithms work, it will be straightforward to demonstrate why.

In keeping with the notation used for Toom-3, let us consider the multiplication of two integers m and n . We construct polynomials $p(x) = m_1x + m_0$ and $q(x) = n_1x + n_0$, recording the value of x at which $p(x) = m$ and $q(x) = n$. In Toom-2, the typical choice of evaluation points is $\{0, 1, \infty\}$, so the matrix-vector multiplications will be as follows for p :

$$\begin{pmatrix} p(0) \\ p(1) \\ p(\infty) \end{pmatrix} = \begin{pmatrix} 0^0 & 0^1 \\ 1^0 & 1^1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} m_0 \\ m_1 \end{pmatrix}$$

And similarly for q . Since the product of two degree-1 polynomials is a degree-2 polynomial, we will need $2 + 1 = 3$ points to specify it:

$$\begin{pmatrix} r(0) \\ r(1) \\ r(\infty) \end{pmatrix} = \begin{pmatrix} 0^0 & 0^1 & 0^2 \\ 1^0 & 1^1 & 1^2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \\ r_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \\ r_2 \end{pmatrix}$$

Computing the inverse of the left-hand matrix in the rightmost part of the equation gives:

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r(0) \\ r(1) \\ r(\infty) \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ r_2 \end{pmatrix}$$

This amounts to the following system of equations:

$$\begin{aligned} r_0 &= r(0) = p(0)q(0) = m_0n_0 \\ r_1 &= r(1) - r(0) - r(\infty) = p(1)q(1) - p(0)q(0) - p(\infty)q(\infty) \\ &= (m_1 + m_0)(n_1 + n_0) - m_0n_0 - m_1n_1 \\ r_2 &= r(\infty) = p(\infty)q(\infty) = m_1n_1 \end{aligned}$$

Therefore $r(x) = (m_1n_1)x^2 + ((m_1 + m_0)(n_1 + n_0) - m_0n_0 - m_1n_1)x + m_0n_0$, which is precisely the formula from section 3.2, merely with different variable names.

3.4 Comparison of algorithm performance bounds

We will use *O*-notation (“big-Oh notation”) to compare the computational complexity of several different algorithms—in other words, to compare how long the algorithms tend to take. Intuitively, *O*-notation allows us to abstract away small details and focus only on the “big picture” of how an algorithm behaves as its inputs grow arbitrarily large. This is not always the most salient aspect of an algorithm’s behavior in a particular context—in fact, the bulk of the new contributions described in chapter 5 deal with determining when the small details become more relevant than the *O* complexity—but it is a valuable tool for analyzing algorithms nonetheless.

Definition 3.1. Let f and g be functions from \mathbb{N} to \mathbb{R}^+ . We say that $f(n) = O(g(n))$ when there exist positive integers c and n_0 such that for every integer $n \geq n_0$, $f(n) \leq cg(n)$. [14, p. 47]

For example, $f(n) = 3n^2 + 4n + 5$ is $O(n^2)$ with $c = 4$ and $n_0 = 5$, since for all $n \geq 5$, $f(n) \leq 4n^2$. But note that although these numbers happen to give a close bound, in the sense that they are the smallest possible choices for $g(n) = n^2$, we could just as well choose different ones: it is equally correct to say that $f(n)$ is $O(n^2)$ because $f(n) \leq 100n^2$ for all $n \geq 9999$, or that $f(n)$ is $O(n^4)$ (feel free to verify this with your own values of c and n_0). According to the definition, there is no rule against including a leading constant or additional terms in $g(n)$ itself and concluding, for example, that $f(n)$ is $O(4n^2 + 1)$, but in practice it is simply not done, because the point of big-O notation in actual use is to choose the *simplest and closest* function $g(n)$, since that will give us the most information. For example, every function which is $O(n^2)$ is $O(n^3)$, but not every function that is $O(n^3)$ is $O(n^2)$, so showing that a function is $O(n^2)$ enables finer classification of its growth rate in comparison to other functions, which is really what we care about in the vast majority of cases.²

²A different formalism, Θ -notation (“theta notation”), allows us to be more precise: $f(n) = \Theta(g(n))$ when $f(n)$ can be bounded both above and below by $cg(n)$ depending on the choice of the constant c [14, p. 44]. Since most literature sources report the complexity of algorithms using *O* rather than Θ even though using Θ would be just as correct and more precise, we follow that convention here when citing them.

In practice, this formal definition is rarely applied directly. Instead, most cases can be handled by a simple heuristic: $f(n)$ is $O(g(n))$ when $g(n)$ is the fastest-growing term in $f(n)$ with the leading constant removed. For example, we can see by inspection that $4n^2 + n$ is $O(n^2)$, or that $3n \log n + 2n$ is $O(n \log n)$.

Logarithms deserve special mention: ordinarily, we need to choose a base whenever we write $\log n$. In computer science we usually mean the binary logarithm $\log_2 n$, and other fields of mathematics often mean $\log_{10} n$. But because conversion to a particular different base b amounts to multiplication by a constant, the specification of a base is not necessary inside of O -notation unless the logarithm occurs inside of some other expression which prevents the suppression of interior coefficients.

Schoolbook multiplication is simple to analyze: to multiply two polynomials of length n requires multiplying each term in the first by each term in the second, for an overall complexity of $O(n^2)$. Karatsuba has a complexity of $O(n^{\log 3}) \approx O(n^{1.59})$ [19, Theorem 8.3]. Toom- k has a complexity of $O(n^{\log_k(2k-1)})$ [7] (note that the base k of the logarithm *does* matter, since it is inside the exponent), which gives the expected result of Toom-2 having the same complexity as Karatsuba (since they are the same algorithm), and gives a complexity of $O(n^{\log_3(5)}) \approx O(n^{1.46})$ for Toom-3.

Since $\lim_{k \rightarrow \infty} \log_k(2k-1) = 1$, it would seem that we could achieve performance arbitrarily close to $O(n)$ by increasing the value of k in Toom- k , splitting the input into more and more different pieces—a very attractive prospect! Unfortunately, the algorithm grows very complicated very quickly as k increases, and the leading coefficient which is suppressed by O -notation quickly becomes unmanageable in practice for inputs of cryptographically relevant sizes.

Other algorithms asymptotically faster than Toom- k for values of k actually used in practice exist, most notably Schönhage and Strassen's $O(n \log n \log \log n)$ algorithm based on the fast Fourier transform [19, Theorem 8.24], but their real-world performance does not exceed Toom- k at the input sizes used in `gf2sliced`.

4. BITSLICING

Usually, when operating on a vector containing many elements, the elements are stored contiguously in memory, and when they are loaded into registers computation, each register stores one element (or part of one element, if the data type is too big to fit). Bitslicing is a technique where the data is instead stored so that each contiguous chunk of memory contains not “all the bits of the i -th element”, but rather, “the i -th bit of each element, concatenated together”: if we think of the elements of the vector as being stacked vertically atop one another, where each row contains one element, then the traditional way of representing the vector stores it row by row, and the bitsliced representation stores it column by column. The term “bitslicing” evokes the sense that we are in essence taking a 1-bit vertical “slice” of the data.

4.1 Overview

A CPU natively operates on data of a certain bit length, these days typically 32 or 64. When performing operations on smaller pieces of data, some (in some cases almost all) of the available space in the computation is wasted: when a byte (8 bits) is stored in a 64-bit register, 56 of the 64 bits are unused. If a register can store 64 bits, and we know that each of our data elements fits into 8 bits, why not put 8 data elements into a single register and compute on them all with a single instruction? For bitwise instructions the only difficulty this poses is in loading the data into the register and retrieving the results, but for instructions such as integer arithmetic that may produce carries, it becomes significantly more complicated to implement this strategy at the software level. Fortunately some instruction sets enable the use of this technique at the CPU level. For example, Intel’s MMX technology allows a 64-bit register to be viewed as an 8-element vector of bytes (8 bits each), a 4-element vector of words (16 bits each), or a 2-element vector of doublewords (32 bits each) [24, p. 229]. In this way, four 16-bit additions can be computed with a single CPU instruction, making effective use of the entire width of the register.

The most fine-grained division it is possible to make is to view a w -bit register as being comprised of w 1-bit elements. This does not require any special intrinsics: the only operations we *can* perform on single bits are bitwise instructions, and therefore we need not be concerned with carries or overflow. This is how we compute with bitsliced data.

Code that is written to take advantage of the ability to process an entire vector at once is often fittingly called “vectorized”. The term is more often applied to code that uses the SIMD instructions for multi-bit elements, but it is technically appropriate for bitsliced code too, since a vector of 1-bit elements is just as much a vector as a vector of, say, 8-bit elements. With the entire vector being operated on at once, vectorized code has a few restrictions: there can be no data-dependent branches, nor data-dependent array accesses. (Those operations can be simulated, but the simulation is slow.) Since performing the same steps for all inputs even in non-bitsliced code protects against timing attacks, finding efficient algorithms that work this way is often a subject of cryptographic research regardless of the underlying data representation. For example, earlier work of this sort on elliptic curve computations was a crucial precursor to the fast bitsliced implementation BBE251 that this thesis builds upon [3].

One of the greatest benefits of bitslicing is actually a property of the vectorized code that this approach forces us to write: the code is resistant by default against side-channel attacks such as cache-timing attacks and branch-prediction attacks. There are no data-dependent array accesses or branches, so it is naturally impossible for an attacker to observe any differences in execution that those might induce, even without the addition of techniques like random masking [3]. Of course, bitsliced algorithms might still be vulnerable to other types of attacks, but it is certainly reassuring to know that one common avenue of potential information leakage during computation is closed off.

However, bitslicing is not a total panacea. As with any specialized implementation technique, converting existing implementations of traditional algorithms into bitsliced implementations takes time, and requires either finding developers who are already familiar with the technique or letting the current developers on a project take time away from their other tasks to learn it. New cryptosystems may be designed with bitslicing in mind from the very beginning, such as Serpent [6], but every new cryptosystem requires intense vetting and study before it can be considered suitable for use, and even then, it may have weaknesses—whether inherent in the underlying mathematics, or present in a particular implementation—that lie undiscovered for a long time. In addition to these two considerations (which apply to any cryptosystem implementation technique), there are some that are specific to bitslicing. Although it is possible to simulate conditional branching in bitsliced code, any algorithm that relies heavily on it will perform poorly (see section 4.2). While bitslicing can speed up batch computations, it may perform worse than even slow traditional implementations if there are not a lot of inputs to process at once; in other words, if it is more important to process a single individual input quickly than to process many inputs quickly on average, bitslicing is likely to be a poor fit, since the program must either wait for more inputs to arrive, or fill the remaining slots with junk data. And finally, bitsliced code can create a huge amount of register pressure: each slice fits into one register, but there are likely to be far more slices than the number of available registers,

so the execution time may be severely affected by a large number of loads and stores.

The first application of bitslicing was in creating a faster implementation of DES, a symmetric encryption algorithm which uses XOR operations, expansion and permutation operations (copying and rearranging bits), and substitution boxes (commonly referred to as “S-boxes”) [5]. DES is particularly well-suited to bitsliced implementation because of its heavy reliance on expansions and permutations: in the traditional representation, these operations are expensive because they require manipulating individual bits (or chunks of bits) within a single register, but in the bitsliced representation, they are quite fast because they do not require manipulating the underlying data at all, but only renaming registers. Since then, progress has been made in applying bitslicing to speed up other block ciphers such as AES [28, 1], designing new systems like Serpent [6] and SHA-3 [38] from the ground up with bitsliced implementation in mind, and of course in speeding up binary field computations.

One more interesting bitslicing-related development in recent years has been the creation of Usuba [32, 31], a domain-specific programming language and accompanying optimizing transpiler that generates efficient bitsliced C code from a high-level description of a cipher. In keeping with the focus on bit-level operations and the intuition of bitslicing-friendly ciphers being essentially “circuits in software”, Usuba is a dataflow language, similar to hardware description languages such as VHDL and Verilog. It also has an intriguing static type theory which ensures that “well-typed programs do always vectorize” [31, Section 1]. The Usuba transpiler (written in OCaml) implements several bitslicing-specific optimizations, paying attention to patterns that are common in bitsliced code and may not be handled well by the optimization passes of the general-purpose C compiler which is ultimately used to compile the generated code [32, Section 4]. Of particular interest is Usuba’s instruction scheduling algorithm¹, which endeavors to minimize the number of register spills (an important task when optimizing bitsliced code, which has an unusually high number of live variables). Although the work in this thesis relies on already existing hand-tuned bitsliced code, Usuba is surely of interest for new projects, and the publications describing it offer much insight into the unique optimization considerations for bitsliced implementations of cryptographic ciphers.

4.2 Simulating data-dependent branching

One of the traits of vectorized code is that all elements must undergo the exact same steps in a computation. For a SIMD register $v = (v_0, v_1, \dots, v_n)$, it is not possible to loop over the elements one by one and operate on them conditionally to do something like “for

¹The implementation is available at <https://github.com/usubalang/usuba/blob/main/src/optimization/scheduler.ml> and is particularly well-commented, such that it should be an enlightening read even for a programmer relatively unfamiliar with OCaml.

each v_i , if $\varphi(v_i)$, set $v_i := f(v_i)$, otherwise set $v_i := g(v_i)$ ” for some test function φ and two functions f and g whose domain and range match the data type of the elements in the SIMD register. However, it is possible to simulate this behavior by constructing a mask vector $m = (m_0, m_1, \dots, m_n)$ where m_i has all bits set to 1 if $\varphi(v_i)$ and all bits set to 0 otherwise, computing $v_f = (f(v_0), f(v_1), \dots, f(v_n))$ and $v_g = (g(v_0), g(v_1), \dots, g(v_n))$, and then computing $(m \wedge v_f) \vee (\neg m \wedge v_g)$.

As an example, consider a 16-bit SIMD register containing four unsigned 4-bit integers, $a = (1111, 1000, 0110, 0011)$. We would like to execute the following data-dependent computation: for each a_i , if $a_i < 1000$, set $a_i := a_i + 1$, otherwise leave it alone. After this computation, the first two elements should remain unchanged, and the last two elements should be updated to 0111 and 0100 respectively.

First, we create the mask vector $m = (m_0, m_1, m_2, m_3)$ such that $m_i = 1111$ when $a_i < 1000$ and $m_i = 0000$ otherwise. In this case we get $m = (0000, 0000, 1111, 1111)$.

Next we compute $a_f = (0000, 1001, 0111, 0100)$. (Actually this can be done first; it doesn't matter.) Since the elements should remain unchanged when the conditional is false, we can immediately write $a_g = a$.

$m \wedge a_f = (0000, 0000, 0111, 0100)$, and $\neg m \wedge a_g = (1111, 1000, 0000, 0000)$, so:

$$\begin{array}{cccc} & 0000 & 0000 & 0111 & 0100 \\ \vee & 1111 & 1000 & 0000 & 0000 \\ \hline & 1111 & 1100 & 0111 & 0100 \end{array}$$

Just as intended, the first two elements have come through unchanged, and the last two have been increased by 1.

The one major potential problem with this approach is that f and g have to be applied to *every* element of the input, even though some of those results will be discarded later. If the functions are quick to compute then this is not such a big issue, but if one or both of them is computationally expensive, or if there are very many places in the code where this technique is used to mimic conditional execution, then it can introduce significant slowdowns.

5. THE FINITE FIELD ARITHMETIC LIBRARY

5.1 Overview

As discussed in chapter 3, different algorithms for polynomial multiplication perform better in practice for different cryptographically relevant input sizes. The objective of `gf2sliced` is to discover the most efficient possible recursive multiplication strategy for a given field size on a particular computer architecture, where the base case for the recursion is the straightline code provided by Bernstein [3]. That is, the most efficient *within reason*—we make several simplifying assumptions along the way, and because execution timings can be affected by other processes on the computer, different runs of the benchmarks may report a different strategy as “the best” for a particular field size. However, the ordering of the strategies by best performance is fairly consistent, with the same few strategies always appearing close together near the top for a given field size on a particular architecture, so not much is lost if a particular run of the benchmarks happens to choose a “best strategy” which tends to appear second or third among multiple runs.

Achieving our optimization goal is a two-step process. First, for a specific field $GF(2^n)$, we generate a collection of potential strategies for multiplication in that field. Then, we generate different versions of the finite field arithmetic library—each one implementing a different one of the generated strategies—and benchmark them to discover which configuration performs optimally. Optimal performance is determined by minimum median cycle count across 2^{16} separate batch multiplications. The data arrives at the finite field layer already in bitsliced form, so the benchmarking process does not take into account the overhead incurred in transposing the data into and out of that form; however, the cost incurred by transposition depends only on the size of the input, not on the particular multiplication strategy used for processing it, so excluding it from the benchmarking process (which always compares results of different multiplication strategies on the *same* field size) is the only logical choice.

The software repository containing the code described in this chapter is available at <https://gitlab.com/nisec/gf2sliced>.

5.2 Strategy generation

We have the following three options available for how to handle multiplication in a particular binary field:

- Three-way recursive split: the WAY30, WAY31, or WAY32 macro is used to emit code that multiplies two $GF(2^n)$ elements by considering each one as a degree-2 polynomial (in another variable, different from the one used to represent the underlying field element—recall the discussion in section 3.3 on Toom-Cook multiplication) and constructing their product as a degree-4 polynomial by interpolation [3, p. 325]. The WAY3 at the beginning indicates that these macros split each input of size n into 3 subproblems of approximately size $\lfloor \frac{n}{3} \rfloor$; the number 0, 1, or 2 is the remainder $n \bmod 3$. Bernstein [3] calls this “five-way recursion” because it constructs the coefficients of a degree-4 polynomial, which has 5 terms; we refer to it as “three-way recursion” or “splitting three ways” in accordance with the names of the macros themselves, or often simply as WAY3, referring to the macro name directly. As shorthand for this choice we write $3\{R\}K\{N\}$, where N is the input size and R is the remainder: for example, 31K334 for a WAY3 split of $n = 334$ (the K refers to Karatsuba).

The subproblem sizes for the WAY3 macros are given in Table 5.1.

- Four-way recursive split: the WAY40, WAY41, WAY42, or WAY43 macro is used to emit code that multiplies two $GF(2^n)$ elements by considering each one as a degree-3 polynomial (same caveat as above) and constructing their degree-6 product by interpolation [3, p. 326]. The naming scheme is the same: these macros split each input of size n into 4 subproblems of approximately size $\lfloor \frac{n}{4} \rfloor$, and the choice of which macro to use is determined by $n \bmod 4$. Similarly as in the WAY3 case, Bernstein calls this “seven-way recursion”, while we call it “four-way recursion” or “splitting four ways” or just WAY4, and as shorthand we write $4\{R\}K\{N\}$, e.g. 40K128 for a WAY4 split of $n = 128$.

The subproblem sizes for the WAY4 macros are also given in Table 5.1.

- Straightline code (no split): if n is within the configured bounds (in our case $5 \leq n \leq 99$), we can use the straightline code from [3] instead of recursing further.¹ We denote this case by $G\{N\}$, e.g. G32 for $n = 32$ (the G refers to GF2, since the function is called e.g. `gf2_mu1_32`).²

¹To save you a trip to the bibliography: the code is available at <http://binary.cr.yip.to/m.html> in simplified form. We use a script to translate the algorithms described in the files on that website into C code.

²Note that the configured lower bound for straightline code also places a lower bound on the input sizes that can be solved with WAY3 or WAY4, since the recursive splits will have subproblems which we must have a way to solve. For example, $n = 18$ can be split into thirds (with resulting subproblem sizes 6 and 8) but not into fourths, since a four-way split would have subproblems of sizes 4 and 5, and 4 is below our lower bound for straightline code. Problems of size $n \geq 20$ can be solved with either kind of recursive split;

	0	1	2	3	
WAY3	n' $n' + 2$	$n' - 1$ $n' + 1$ $n' + 2$	n' $n' + 1$ $n' + 2$	–	where $n' = \lfloor \frac{n}{3} \rfloor$
WAY4	n'	n'	n' $n' + 1$	n' $n' + 1$	where $n' = \lfloor \frac{n}{4} \rfloor$

Table 5.1. Subproblem sizes of WAY macros

Building a multiplication strategy starts with the choice of which algorithm to use at the uppermost level to handle the initial input size. In practice, since the smallest field we generate a library for is $GF(2^{113})$ and 113 is larger than 99, we always start with at least one level of recursion. This choice uniquely determines what sizes of subproblems will need to be solved: for example, if the initial input size is 251 (i.e. we are multiplying elements of $GF(2^{251})$), we can either begin with a WAY3 split, which will necessitate solving subproblems of sizes 83, 84, and 85, or we can begin with a WAY4 split, which will necessitate solving subproblems of sizes 62 and 63. Then, for each of those subproblems, we choose which algorithm to use to solve it; if one or more of those choices is a recursive algorithm, then we repeat this process for the new subproblems which that call has introduced; and so on, until finally we reach a point where there are no more new subproblems that need to be solved, i.e. every path through the tree of recursive calls ends in a leaf node of a straightline function. For example, the following strategy is complete:

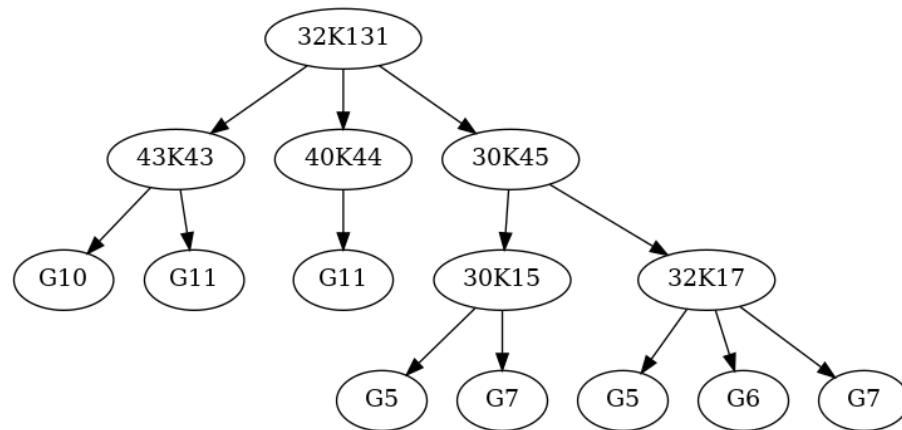


Figure 5.1. One possible strategy for multiplying elements of $GF(2^{131})$

Generation of strategies for a given field size is done by recursively exploring the possible sequences of choices and building a list of all the possible trees that could result. A

problems of size $n \in \{15, 17, 18, 19\}$ can be solved with WAY3 but not WAY4; problems of size $n < 15$ or $n = 16$ must be solved with straightline code (a three-way split of 16 would have a subproblem of size 4). We take this into account while generating strategies, and never attempt a recursive split of a subproblem that is too small to accommodate it.

reader familiar with recursive algorithms may be getting nervous by this point about the combinatorial explosion which happens here, and indeed they would be correct to do so: without some additional restrictions on the search space, the number of strategies produced becomes impractically large far before we ever approach our desired largest field size of 571. To handle this, we apply several pruning rules, thereby cutting the generated set of strategies down to a manageable size without (as far as we believe) discarding any that would be a significant improvement over the best one that remains. The ones we eliminate are:

- Strategies where the same problem size is solved differently in different recursive calls. For example, if two different recursive calls both require solving a subproblem of size 60, they must both use 30K60, or both use 40K60, or both use G60.
- Strategies that do not have a “strict recursion threshold”: strategies where there is crossover between the range of subproblem sizes solved recursively and the range of subproblem sizes solved with straightline multiplication. For example, if the largest subproblem solved with straightline code is $n = 50$ (equivalently, all problem sizes 51 or larger are solved recursively), then all subproblems smaller than that (49 or below) must also be solved with straightline code.
- Strategies with “mixed multiplication”: where a single recursive call uses further recursion for one or more of its subproblems but also uses straightline multiplication for one or more of the others. For example, a WAY3 split of 113 requires solving subproblems of sizes 39, 38, and 37; those can all be solved recursively (although it is possible to use WAY3 for some of them and WAY4 for others), or they can all be solved with straightline code, but it is not possible to “mix and match” and e.g. recurse on 39 and 38 but use straightline code for 37. (Note, however, that as in the example above, it is acceptable for a single *layer* of the recursion tree to have both recursive and straightline calls—G11 and 30K15 both occur at depth 2—as long as each *individual node* does not mix and match among its direct children.)

The first constraint is a handy one, since it lets us unambiguously represent each tree merely by the set of its nodes. The tree can be “unfolded” again by combining this information with the rules for which subproblem sizes are needed by a given recursive split. We find it convenient to sort the nodes in descending order by input size when displaying them this way. The strategy shown in Figure 5.1 can be represented compactly like this³:

(32K131, 30K45, 40K44, 43K43, 32K17, 30K15, G11, G10, G7, G6, G5)

Now, to gain a better grasp on how the process unfolds as we generate the entire set of

³A quirk of the initial implementation in Python leaves us outputting these as tuples, rather than sets or lists, hence the use of parentheses rather than square or curly brackets. It is useful in the benchmarking phase: tuples—unlike lists—are hashable [21], so we can build a dictionary with strategies as keys and cycle counts as values.

strategies for a given field size, we will work through the strategy generation for $n = 58$. (As mentioned above, the smallest field size we generate a library for is 113; however, 58 appears as a subproblem for some larger fields, and it is chosen as the example size here because it is large enough to exhibit most of the behavior we care about but small enough to be presented in full in a reasonable amount of page space.)

- 58 is small enough that it can be solved directly with straightline code, so G58 is an option.
- 58 can be split 3 ways, to 31K58. This will require solving the following subproblems: 21, 20, and 18.
 - All of these problems can be solved with straightline code, so one option is to finish immediately with G21, G20, G18.
 - All of these problems can be solved recursively: 21 can be split 3 ways or 4 ways, as can 20, but as discussed above, 18 can only be split 3 ways. Therefore there are 4 possible ways to continue from here: 2 choices for 21, times 2 choices for 20, times 1 choice for 18.
 - * 30K21, 32K20, 30K18: these three problems combined require solving subproblems of size 9, 8, 7, and 6, which are all too small for further recursion, so we finish with G9, G8, G7, G6.
 - * 30K21, 40K20, 30K18: similarly, we finish with G9, G8, G7, G6, G5.
 - * 41K21, 32K20, 30K18: similarly, we finish with G8, G7, G6, G5.
 - * 41K21, 40K20, 30K18: similarly, we finish with G8, G6, G5.
- 58 can be split 4 ways, to 42K58. This will require solving the following subproblems: 15 and 14.
 - 15 is large enough to admit further recursion, but 14 is not, so according to the “no mixed multiplication” constraint, all we can do from here is immediately finish with G15, G14.

So the final set of generated strategies is:

(G58)

(31K58, G21, G20, G18)

(31K58, 30K21, 32K20, 30K18, G9, G8, G7, G6)

(31K58, 30K21, 40K20, 30K18, G9, G8, G7, G6, G5)

(31K58, 41K21, 32K20, 30K18, G8, G7, G6, G5)

(31K58, 41K21, 40K20, 30K18, G8, G6, G5)

(42K58, G15, G14)

5.3 Code generation and benchmarking

For every strategy that we generate, we need to compile the field arithmetic library with that strategy and test it, so that we can determine which one has the best performance.

The first step is to write out a C header file, the inclusion of which will configure the library to use the particular strategy under consideration. The transformation of a strategy into a header file is straightforward: we define a constant `GF2_M` for the current field size; include the necessary utility file `gf2_utl_{N}.c` which defines a macro for modular reduction and functions for squaring and inversion; include the needed straightline multiplication files; and then call the `WAY` macros corresponding to our choice of recursive splits. Everything except the initial definition of `GF2_M` is gated behind the `CONFIG_GF2_SPLIT_` flag, which is set in the main library file `gf2.c` to indicate that the results of our strategy generation should indeed be used (rather than a predefined/hardcoded configuration from some other source besides our strategy generation tool). For example, the strategy (43K283, 43K71, 42K70, G18, G17) generates the following `config.h`:

```
#define GF2_M 283

#ifdef CONFIG_GF2_SPLIT_
#include "gf2_utl_283.c"

#include "gf2_mul/gf2_mul_17.c"
#include "gf2_mul/gf2_mul_18.c"

WAY42(70, gf2_mul_17, gf2_mul_18)
WAY43(71, gf2_mul_17, gf2_mul_18)
WAY43(283, karatmult70, karatmult71)
#undef CONFIG_GF2_SPLIT_
#endif /* CONFIG_GF2_SPLIT_ */
```

Figure 5.2. A configuration header file for $GF(2^{283})$

The code is then compiled with that configuration, and the resulting library is linked to the benchmarking harness, which can perform one or both of the following tasks:

- Benchmarking of the field arithmetic library: cycle counts for 2^{16} batched multiplications of pseudorandom field elements are collected (using the `RDTSC` instruction, with the `CPUID` instruction inserted to prevent `RDTSC` reporting inaccurate cycle counts due to out-of-order execution—for details on this technique, see Paolini [37]), and the median is reported.
- Ad-hoc testing of the field arithmetic library: a batch of pseudorandom elements

is inverted, then multiplied by the computed inverses, and the resulting batch is checked to ensure that every element in it is 1.

During benchmarking, we use both of these features, to ensure that we not only collect performance data but also have a chance to catch any unexpected errors or bugs. The process of compiling and benchmarking each configuration is automated with a Python script, and as that script makes its way through the generated strategies for the specified field size, it stores each strategy's median cycle count. Once all strategies have been tested, the script writes out the configuration header file for the one with the lowest cycle count, and any errors (i.e. cases where the inversion test failed) are reported by printing out the strategy configuration in which they occurred. The list of generated strategies and their cycle counts is also written out as a plain text file so that further analysis can be performed if desired, either "by inspection" or with utilities like `awk`.

5.4 History

During the writing of this thesis, several interrelated bugs were discovered in the original Python implementation of the algorithm for generating splitting strategies. Although the bugs did not result in the generation of any invalid strategies, they did result in some valid strategies not being generated for certain field sizes, and in the case where any strategies were missed, they were missed nondeterministically: different runs of the generator could emit different subsets of the set of all valid strategies.⁴ The nature of the bugs were that if multiple strategies for the same field size used the same set of subproblem sizes as each other and had the same cutoff for when to switch to straightline multiplication, only one of the strategies would be included in the final output of the generator. This happened because two strategies were considered equal when they were equal elementwise, and the implementation of elementwise equality only checked that the value (i.e. the (sub)problem size) was the same and that the elements were either both straightline or both recursive: it did not check to ensure that two recursive splits of the same value used the same `WAY` macro.⁵ As an example, consider the following two valid strategies for $n = 220$:

(40K220, 31K55, 40K20, 31K19, 32K17, G8, G7, G6, G5)

⁴If this sounds like a bizarre way for even a very buggy implementation of a theoretically completely deterministic algorithm to behave, rest assured that the author felt the same way when it was discovered! It lay undiscovered for a long time because the generator behaved deterministically when repeatedly called from the same Python interpreter session, which is how it was usually tested during development. The behavior turned out to be due to the interaction between Python's string hashing method (it salts them, and the seed value is different across different invocations of the Python interpreter but remains the same within each session) and the fact that the iteration order of a `set` can depend on the hashes of the items.

⁵This was a simplifying assumption based on the observation that any *individual* value would require different subproblem sizes for a three-way split versus a four-way split, so the overall strategies containing either choice were expected to be distinguishable from each other because the sets of subproblem sizes used in them would be different. However, this failed to account for the situation where other subproblem splits in the strategy already cover all of the subproblem sizes which would be introduced by either a three-way or four-way split of a given value.

(40K220, 31K55, 32K20, 31K19, 32K17, G8, G7, G6, G5)

One uses 40K20 and the other uses 32K20. In isolation, these would be distinguishable from each other as strategies because the former has a subproblem of size 5 and the latter has subproblems of sizes 6, 7, and 8. But the overall strategies both contain 31K19 (subproblem sizes 5, 7, and 8) and 32K17 (subproblem sizes 5, 6, and 7), so regardless of what choice is made about how to split the subproblem $n = 20$, the resulting overall strategy will contain subproblems of sizes 5, 6, 7, and 8.

Preliminary, “obvious” bugfixes to the existing code were able to at least make strategy generation deterministic, but the generated set of strategies remained incomplete for some field sizes, and, even worse, the code became so slow and memory-intensive that strategies for some field sizes could take over an hour to generate, and in particular the strategies for $GF(2^{568})$ could not be generated at all: the process would run for several hours and then get terminated by the Linux kernel’s out-of-memory killer.

A standalone proof-of-concept of a corrected implementation was prototyped in Haskell (for ease of working with and reasoning about the recursive data structures involved), then translated into Python to avoid adding additional dependencies to the project. As of March 2023 it has not yet been integrated into the main codebase, and we have not yet rerun the benchmarks under appropriately controlled conditions to obtain comparable data on the cycle counts of the best strategies from the full set of strategies. However, even with some valid strategies missing, the original benchmarking data showed sufficient improvement over existing results in the literature to merit publication, and since the best few strategies found by the original version of the tool for a given field size tended to be very close in performance to each other anyway, it is unlikely that significantly better strategies were being missed in any systematic way. The main advantage of the new implementation (other than maintainability and of course correctness) is that it generates the strategies significantly faster, leading to reduced build times at larger field sizes.

6. RESULTS

`gf2sliced` was used to generate and benchmark binary field multiplication strategies for the full set of field sizes in the range $n \in [113, 571]$ on four different platforms [11]. Cycle counts were normalized by dividing them by the SIMD register width so that the performance per each individual multiplication could be compared with non-bit-sliced implementations. Since it is common for results to be reported for the field size $n = 251$ in the literature on this topic, that field size was used as the main point of comparison with prior work. It was established that on machines with a 512-bit SIMD register width (using the AVX-512 instruction set), the best strategies generated by `gf2sliced` outperformed almost all existing implementations, and on less powerful machines using the more widespread 256-bit width AVX2 instruction set, the best strategies generated by `gf2sliced` still compared favorably to previous results. The one previously existing implementation we found which was as efficient as `gf2sliced` even at the 512-bit width and significantly outperformed it at the 256-bit width was the one in [39], which takes advantage of the ARMv8 PMULL operation to perform 64-bit polynomial multiplication in a single instruction. For details see appendix A.

In [42], the benchmarks were run again on a different machine for the same subset of field sizes for which explicit cycle counts were reported in [11]. Whereas [11] focused on proving that the performance of code generated by `gf2sliced` was competitive with or even superior to other unrelated implementations already available, [42] focused on analyzing the distribution of cycle counts for the collection of strategies generated for those field sizes in more detail. The cycle counts were reported for a complete batch of elements, rather than being normalized by the SIMD register width, so the raw numbers are not directly comparable to those previously published. However, knowing that the processor used for that round of benchmarks supports the 256-bit width AVX2 instruction set (but not the 512-bit width AVX-512), we can normalize the cycle counts ourselves. Table 6.1 shows the normalized cycle counts from [11] for AVX-512 (Intel Xeon Silver 4116), AVX2 (Intel Core i5-6500), and AVX2-AMD (Ryzen 7 2700X), and from [42] for AVX2 (Intel Core i5-7200U). The normalized cycle counts on AVX2 are similar in both cases.

n	113	131	163	191	193	233	239	251	283	359	409	431	571
AVX-512	13	16	22	30	31	44	44	51	59	82	99	104	170
AVX2 [11]	24	30	43	54	55	84	77	91	121	166	216	214	351
AVX2 [42]	28	36	57	66	66	101	93	110	133	186	238	251	398
AVX2-AMD	37	47	67	83	89	114	116	132	153	224	274	300	459

Table 6.1. Binary field multiplication performance in normalized CPU cycles

As discussed in section 5.4, these benchmarking runs inadvertently excluded some valid strategies for some field sizes due to bugs in the initial implementation of the strategy generation algorithm. Comparing the number of strategies benchmarked for each selected field size in [42] to the number of strategies generated for that field size by the fixed implementation, the field sizes for which some strategies were missed are as follows. (In all cases we subtract the number of strategies, if any, for which the generated code did not produce correct results. See later in the chapter for a discussion of which strategies produced errors.)

- 283: 315 benchmarked out of 331 total, or 4.8% missed
- 409: 322 benchmarked out of 345 total, or 6.7% missed
- 431: 356 benchmarked out of 371 total, or 4.0% missed
- 571: 1312 benchmarked out of 2717 total, or 51.7% missed (no, that's not a typo!)

The fact that the observed cycle count of the best-performing strategy grows more or less linearly as a function of the field size with no extreme outliers (see appendix A, Fig. 2), when some but not all field sizes had valid strategies missing from the set of strategies which were benchmarked for them, supports the hypothesis of section 5.4 that there was no field size at which the bugs caused all high-performing strategies to be excluded.

For this thesis, *all* generated strategies from the fixed implementation were tested for all field sizes to ensure that no information was missed about which strategies produce errors. The cycle counts from these test runs are not meaningful as performance measurements, because they were noticeably affected by running some of the tests in parallel; since the performance characteristics of the best strategies have already been well-explored in prior publications, we focus here on analyzing the nature of the generated strategies themselves. In particular, we investigate how their total number depends on the field size and on the upper limit of which subproblems can be solved with straightline multiplication, and we finally uncover which aspect determines whether a given strategy's generated code will pass the inversion test or not.

As shown in Figure 6.1, the relationship between the field size and the number of generated strategies exhibits some interesting structure in and of itself. The points clearly split

into two bands as n grows, and the rate of growth changes sharply around $GF(2^{250})$ and again at around $GF(2^{465})$. The upper band consists of strategy counts for which n is not divisible by 3, and the lower band consists of strategy counts for which n is divisible by 3. This is because the WAY31 and WAY32 macros each require solving 3 subproblems, whereas WAY30 and all four WAY4 macros only require solving 1 or 2.

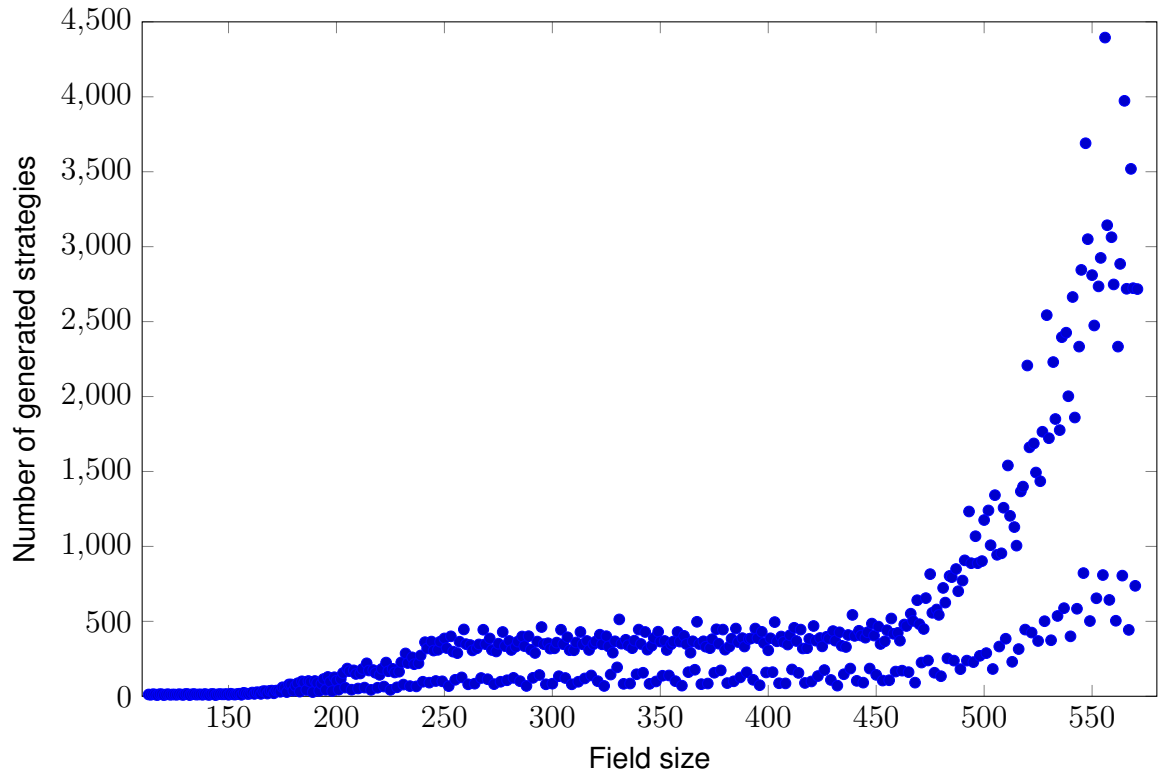


Figure 6.1. Strategy counts with straightline upper limit 99

One idea Figure 6.1 inspires for how to speed up the benchmarking process is to use the benchmarking results for fields smaller than $GF(2^{465})$ or thereabouts to preemptively prune known-inefficient strategies from the strategy generation process for larger fields. The soundness of making this change would depend on the truth of the simplifying assumption that the performance characteristics of the strategies for a certain field size are the same—or at least similar enough—regardless of whether they appear at the uppermost level of recursion or as a subproblem within a strategy for a larger field. This could be a fruitful area for further study: if it turns out to be true, then optimal or near-optimal strategies for larger field sizes could be discovered in minutes instead of hours, if smaller field sizes have already been benchmarked.

In [42, p. 36] it was suggested that since the largest field size for which any of the best-performing strategies used straightline code was $GF(2^{23})$, the upper limit for straightline code could be lowered from 99 to 23 without eliminating the best strategy. As shown in Figure 6.2, this does reduce the number of generated strategies, but only slightly. Since 23 is still large enough to be solved with either a three-way split or a four-way split, and the

“no mixed multiplication” constraint ensures that there is never more than one way to use straightline code in a given call’s subproblems but places no restraints on mixing three-way and four-way splits for the subproblems, the combinatorial nature of the problem means that for each subtree eliminated this way, there remain 2, 4, or 8 ways to choose recursive calls for its next layer (depending on whether the recursive split in its root node produces 1, 2, or 3 subproblems), and unless the subproblems in that layer are already quite small, they may each admit multiple strategies for their own subproblems as well. In other words, this change is almost certainly harmless, but the benefits are minimal.

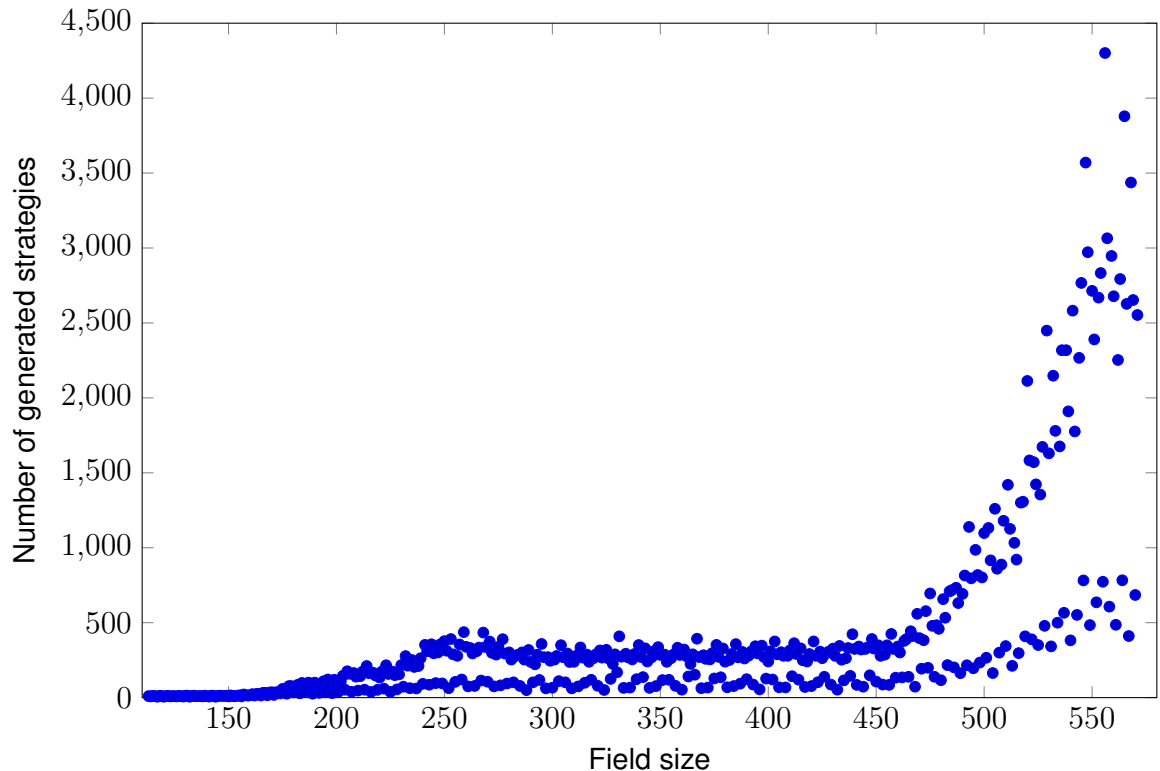


Figure 6.2. Strategy counts with straightline upper limit 23

Another question which has now been at least partially answered is the question of why some field sizes have many strategies which fail the inversion test while other field sizes have no failing strategies at all. After analyzing the structure of the strategies with errors, it was discovered that the ones with errors are exactly those which include 31K19. It is not yet known why the generated code for `WAY31(19, gf2_mul_5, gf2_mul_7, gf2_mul_8)` produces incorrect results, but at least now the source of the problem is clear. As a stopgap measure while this bug remains unfixed, we could configure the generator to prohibit strategies containing 31K19 from being emitted at all. For some field sizes the impact would be significant: there are 23 field sizes for which more than 1000 of the generated strategies contain 31K19. Figure 6.3 shows the number of strategies with errors for each field size, and Figure 6.4 shows what percentage of the total strategies for each field size (with straightline upper limit 99) have errors.

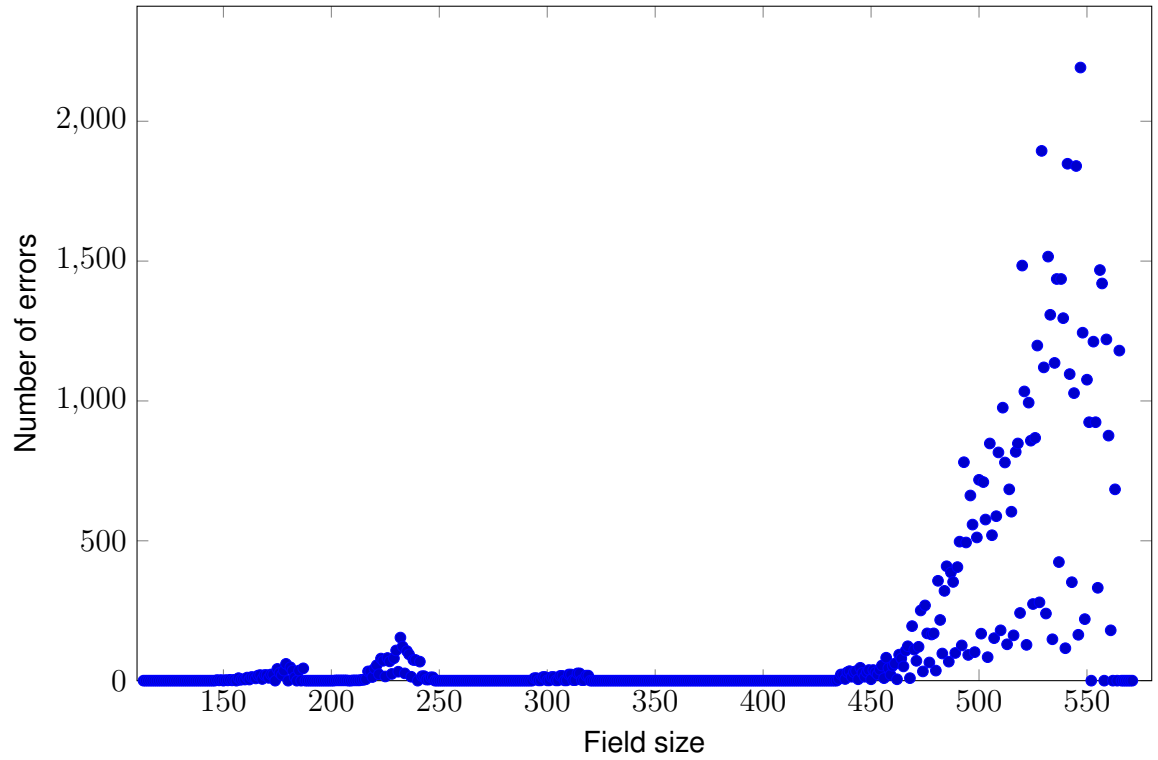


Figure 6.3. Strategy counts containing 31K19

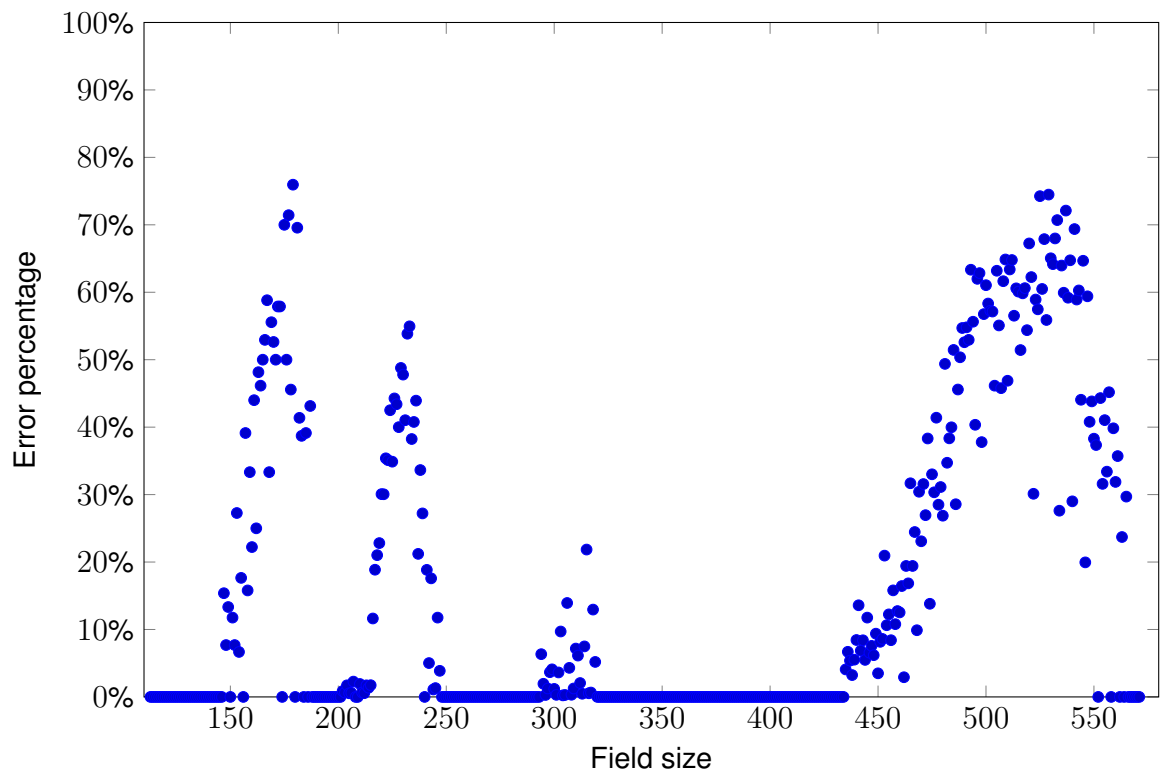


Figure 6.4. Percentage of strategies with errors (straightline upper limit 99)

7. CONCLUSION

Bitsliced computation enables calculations to be performed faster on average when there tend to be enough inputs available at once that they can be processed in batches, for example on a high-traffic server which must perform key negotiation with many clients. The only algorithms amenable to bitsliced implementation are those which use little or no conditional branching or data-dependent array indexing, but those are desirable properties for cryptographic algorithms to have anyway since they increase resistance to side-channel attacks. Despite the apparent “special case” nature of the point at infinity \mathcal{O} , some elliptic curves admit point addition laws with no branching [4], making them bitslicing-friendly.

Many different recursive multiplication algorithms are known for the fields over which elliptic curves are defined, and their asymptotic performance is generally well-understood, but the elements we work with in elliptic curve cryptography over binary fields are not large enough for the most asymptotically efficient algorithm to perform best in practice, so a combination of approaches is needed. Automating the generation and benchmarking of various recursion strategies allows our tool to discover the best recursion strategies for a specific computer architecture at compile time, without needing manual tuning. A rewrite of the strategy generation algorithm eliminated some bugs that had been preventing the full set of valid strategies from being generated, and made the generation process much faster and less memory-intensive as well; however, many improvements still remain to be made in the ergonomics of the build tooling for the overall software package.

The most important next step for future work is to understand why the code generated by the WAY31 macro produces incorrect results for $n = 19$, and to fix it. Once this is done, all of the generated strategies should pass the inversion test. A logical next step for obtaining further improvement in the efficiency of the benchmarking process itself could be to use the benchmarking results for smaller field sizes to preemptively prune strategies which have already been discovered to be inefficient; the soundness of this simplification would have to be experimentally tested. Finally, analysis of the structure of the best- and worst-performing strategies may also be fruitful, to develop additional static constraints which could further reduce the number of strategies which must be benchmarked to find the optimal ones.

REFERENCES

- [1] Alexandre Adomnicai and Thomas Peyrin. “Fixslicing AES-like Ciphers: New bit-sliced AES speed records on ARM-Cortex M and RISC-V”. en. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), pp. 402–425. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i1.402-425. URL: <https://tches.iacr.org/index.php/TCHES/article/view/8739>.
- [2] Michael Artin. *Algebra*. 2nd ed. Boston, MA: Pearson Education, 2011. ISBN: 978-0-13-241377-0.
- [3] Daniel J. Bernstein. “Batch Binary Edwards”. en. In: *Advances in Cryptology - CRYPTO 2009*. Ed. by Shai Halevi. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 317–336. ISBN: 978-3-642-03356-8. DOI: 10.1007/978-3-642-03356-8_19.
- [4] Daniel J. Bernstein and Tanja Lange. “Faster Addition and Doubling on Elliptic Curves”. en. In: *Advances in Cryptology – ASIACRYPT 2007*. Ed. by Kaoru Kurosawa. Vol. 4833. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 29–50. ISBN: 978-3-540-76899-9. DOI: 10.1007/978-3-540-76900-2_3. URL: http://link.springer.com/10.1007/978-3-540-76900-2_3.
- [5] Eli Biham. *A Fast New DES Implementation in Software*. CS Technion report CS0891. 1997. URL: <http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi/1997/CS/CS0891>.
- [6] Eli Biham, Ross Anderson, and Lars Knudsen. “Serpent: A New Block Cipher Proposal”. en. In: *Fast Software Encryption*. Ed. by Serge Vaudenay. Vol. 1372. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 222–238. ISBN: 978-3-540-64265-7. DOI: 10.1007/3-540-69710-1_15. URL: http://link.springer.com/10.1007/3-540-69710-1_15.
- [7] Marco Bodrato. “Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0”. In: *Proceedings of the 1st International Workshop on Arithmetic of Finite Fields*. WAIFI ’07. Madrid, Spain: Springer-Verlag, 2007, pp. 116–133. ISBN: 9783540730736. DOI: 10.1007/978-3-540-73074-3_10. URL: https://doi.org/10.1007/978-3-540-73074-3_10.
- [8] Richard P. Brent et al. “Faster Multiplication in $GF(2)[x]$ ”. In: *Proceedings of the 8th International Conference on Algorithmic Number Theory*. ANTS-VIII’08. Banff, Canada: Springer-Verlag, 2008, pp. 153–166. ISBN: 3540794557.

- [9] Aiden A. Bruen, Mario Forcinito, and James M. McQuillan. *Cryptography, information theory, and error-correction: a handbook for the 21st century*. Second edition. Hoboken, NJ: Wiley, 2021. ISBN: 978-1-119-58239-7.
- [10] Billy Bob Brumley and Dan Page. “Bit-Sliced Binary Normal Basis Multiplication”. In: *2011 IEEE 20th Symposium on Computer Arithmetic*. Tuebingen, Germany: IEEE, July 2011, pp. 205–212. ISBN: 978-1-4244-9457-6. DOI: 10.1109/ARITH.2011.36. URL: <http://ieeexplore.ieee.org/document/5992128/>.
- [11] Billy Bob Brumley et al. “Batch Binary Weierstrass”. In: *Progress in Cryptology – LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2–4, 2019, Proceedings*. Santiago de Chile, Chile: Springer-Verlag, 2019, pp. 364–384. ISBN: 978-3-030-30529-1. DOI: 10.1007/978-3-030-30530-7_18. URL: https://doi.org/10.1007/978-3-030-30530-7_18.
- [12] Lily Chen et al. *Recommendations for Discrete Logarithm-Based Cryptography: Elliptic Curve Domain Parameters*. NIST Special Publication (SP) 800-186 (Draft). National Institute of Standards and Technology, 2019. DOI: 10.6028/NIST.SP.800-186-draft. URL: <https://csrc.nist.gov/publications/detail/sp/800-186/draft>.
- [13] Stephen A. Cook. “On the minimum computation time of functions”. PhD thesis. 1966. URL: <https://cr.yp.to/bib/entries.html#1966/cook>.
- [14] Thomas H. Cormen et al. *Introduction to algorithms*. 3rd ed. Cambridge, Mass: MIT Press, 2009. ISBN: 978-0-262-03384-8.
- [15] Whitfield Diffie and Martin E. Hellman. “New directions in cryptography”. In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638. URL: <http://ieeexplore.ieee.org/document/1055638/>.
- [16] David Steven Dummit and Richard M. Foote. *Abstract algebra*. 3rd ed. Hoboken, NJ: Wiley, 2004. ISBN: 978-0-471-43334-7.
- [17] Harold M. Edwards. *Galois theory*. Corr. 3. printing. Graduate Texts in Mathematics. New York Heidelberg: Springer, 1998. ISBN: 978-0-387-90980-6.
- [18] Thomas A. Garrity. *All the math you missed: but need to know for graduate school*. Second edition. Cambridge, New York: Cambridge University Press, 2021. ISBN: 978-1-316-51840-3. DOI: 10.1017/9781108992879.
- [19] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. 3rd ed. Cambridge, 2013. ISBN: 978-1-107-03903-2.
- [20] D. Gillmor. *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)*. Tech. rep. Aug. 2016. DOI: 10.17487/rfc7919. URL: <https://doi.org/10.17487/rfc7919>.
- [21] *Glossary — Python 3.11.3 Documentation*. Accessed on 12.05.2023. URL: <https://docs.python.org/3.11/glossary.html#term-hashable>.

- [22] Darrel R. Hankerson, Scott A. Vanstone, and A. J. Menezes. *Guide to elliptic curve cryptography*. en. Springer, 2003. ISBN: 978-0-387-95273-4.
- [23] M.E. Hellman. “An overview of public key cryptography”. In: *IEEE Communications Magazine* 40.5 (May 2002), pp. 42–49. ISSN: 1558-1896. DOI: 10.1109/MCOM.2002.1006971.
- [24] *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture*. en. 2016. URL: <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf>.
- [25] Anatolii Alexeevich Karatsuba. “The complexity of computations”. In: *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation* 211 (1995), pp. 169–183.
- [26] T. Kivinen and M. Kojo. *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*. Tech. rep. May 2003. DOI: 10.17487/rfc3526. URL: <https://doi.org/10.17487/rfc3526>.
- [27] Neal Koblitz. “Elliptic curve cryptosystems”. In: *Mathematics of Computation* 48.177 (1987), pp. 203–209. ISSN: 0025-5718, 1088-6842. DOI: 10.1090/S0025-5718-1987-0866109-5. URL: <https://www.ams.org/mcom/1987-48-177/S0025-5718-1987-0866109-5/>.
- [28] Robert Könighofer. “A Fast and Cache-Timing Resistant Implementation of the AES”. en. In: *Topics in Cryptology – CT-RSA 2008*. Ed. by Tal Malkin. Vol. 4964. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 187–202. ISBN: 978-3-540-79262-8. DOI: 10.1007/978-3-540-79263-5_12. URL: http://link.springer.com/10.1007/978-3-540-79263-5_12.
- [29] Rudolf Lidl and Harald Niederreiter. *Introduction to finite fields and their applications*. Rev. ed. Cambridge, New York: Cambridge University Press, 1994. ISBN: 978-0-521-46094-1.
- [30] Mitsuru Matsui and Junko Nakajima. “On the Power of Bitslice Implementation on Intel Core2 Processor”. In: *Cryptographic Hardware and Embedded Systems - CHES 2007*. Ed. by Pascal Paillier and Ingrid Verbauwhede. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 121–134. ISBN: 978-3-540-74735-2. DOI: 10.1007/978-3-540-74735-2_9.
- [31] Darius Mercadier and Pierre-Évariste Dagand. “Usuba: high-throughput and constant-time ciphers, by construction”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Phoenix, AZ, USA: ACM, June 2019, pp. 157–173. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314636. URL: <https://dl.acm.org/doi/10.1145/3314221.3314636>.
- [32] Darius Mercadier et al. “Usuba: Optimizing & Trustworthy Bitslicing Compiler”. In: *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*. Vienna, Austria: ACM, Feb. 2018, pp. 1–8. ISBN: 978-1-4503-5646-6.

- DOI: 10.1145/3178433.3178437. URL: <https://dl.acm.org/doi/10.1145/3178433.3178437>.
- [33] Ralph C. Merkle. “Secure communications over insecure channels”. In: *Communications of the ACM* 21.4 (1978), pp. 294–299. ISSN: 0001-0782. DOI: 10.1145/359460.359473. URL: <https://dl.acm.org/doi/10.1145/359460.359473>.
- [34] Victor S. Miller. “Use of Elliptic Curves in Cryptography”. In: *Advances in Cryptology — CRYPTO ’85 Proceedings*. Ed. by Hugh C. Williams. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1986, pp. 417–426. ISBN: 978-3-540-39799-1. DOI: 10.1007/3-540-39799-X_31.
- [35] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information*. 10th anniversary ed. Cambridge, New York: Cambridge University Press, 2010. ISBN: 978-1-107-00217-3.
- [36] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Heidelberg, New York: Springer, 2010. ISBN: 978-3-642-04100-6.
- [37] Gabriele Paolini. *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. White Paper 324264–001. Sept. 2010. URL: <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf>.
- [38] Peter Schwabe, Bo-Yin Yang, and Shang-Yi Yang. “SHA-3 on ARM11 processors”. In: *Progress in Cryptology – AFRICACRYPT 2012*. Ed. by Aikaterini Mitrokotsa and Serge Vaudenay. Vol. 7374. Springer-Verlag Berlin Heidelberg, 2012, pp. 324–341. URL: <http://cryptojedi.org/papers/#sha3arm>.
- [39] Hwajeong Seo et al. “Binary field multiplication on ARMv8”. In: *Security and Communication Networks* 9.13 (2016), pp. 2051–2058. DOI: <https://doi.org/10.1002/sec.1462>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1462>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1462>.
- [40] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. en. Vol. 106. Graduate Texts in Mathematics. New York, NY: Springer New York, 2009. ISBN: 978-0-387-09493-9. DOI: 10.1007/978-0-387-09494-6. URL: <http://link.springer.com/10.1007/978-0-387-09494-6>.
- [41] Joseph H. Silverman and John T. Tate. *Rational Points on Elliptic Curves*. en. Undergraduate Texts in Mathematics. Springer International Publishing, 2015. ISBN: 978-3-319-18587-3. DOI: 10.1007/978-3-319-18588-0. URL: <https://link.springer.com/10.1007/978-3-319-18588-0>.
- [42] Kide Vuojärvi. “Underlying Mathematics of Bitsliced Polynomial Multiplication”. Master’s thesis. Tampere University, 2021. URL: <https://trepo.tuni.fi/handle/10024/135226>.
- [43] Peter Wayner. “British Document Outlines Early Encryption Discovery”. In: *The New York Times* (Dec. 1997). URL: <https://archive.nytimes.com/www.nytimes.com/library/cyber/week/122497encrypt.html>.

APPENDIX A: BATCH BINARY WEIERSTRASS

Batch Binary Weierstrass

Billy Bob Brumley, Sohaib ul Hassan, Alex Shaindlin, Nicola Tuveri, and
Kide Vuojärvi

Tampere University, Tampere, Finland

{billy.brumley,n.sohaibulhassan,chloenatasha.shaindlin}@tuni.fi
{nicola.tuveri,kide.vuojarvi}@tuni.fi

Abstract. Bitslicing is a programming technique that offers several attractive features, such as timing attack resistance, high amortized performance in batch computation, and architecture independence. On the symmetric crypto side, this technique sees wide real-world deployment, in particular for block ciphers with naturally parallel modes. However, the asymmetric side lags in application, seemingly due to the rigidity of the batch computation requirement. In this paper, we build on existing bitsliced binary field arithmetic results to develop a tool that optimizes performance of binary fields at any size on a given architecture. We then provide an ECC layer, with support for arbitrary binary curves. Finally, we integrate into our novel dynamic OpenSSL engine, transparently exposing the batch results to the OpenSSL library and linking applications to achieve significant performance and security gains for key pair generation, ECDSA signing, and (half of) ECDH across a wide range of curves, both standardized and non-standard.

Keywords: applied cryptography · public key cryptography · elliptic curve cryptography · software implementation · batching · bitslicing · OpenSSL

1 Introduction

The use of Elliptic Curve Cryptography (ECC) was first suggested in 1985, independently by Miller [23] and Koblitz [19]. Due to the fast group law on elliptic curves and the absence of known sub-exponential attacks, ECC has since gathered momentum as an alternative to popular asymmetric cryptosystems like RSA and DSA, as it can provide the same security level with keys that are much shorter, gaining both in computational efficiency and bandwidth consumption.

A prevalent assumption is that for software implementations, curves over large characteristic prime fields are generally more efficient than their binary field counterparts, and that, vice versa, the opposite applies for hardware implementations, i.e., that ECC scalar point multiplication (usually the most costly operation in ECC cryptosystems) is much faster over binary extension fields.

As we will detail in the following sections, the first half of the above assumption has been repeatedly challenged, especially considering the effect of alternative coordinate representations and the advances in the manufacturing processes

and design of general purpose CPUs, e.g. with the introduction of dedicated units for carry-less multiplication and the widespread adoption of vector processing (a.k.a. SIMD: Single Instruction, Multiple Data) in popular Instruction Set Architectures (ISA).

In this work, we focus on bitsliced binary field arithmetic as a strategy to take advantage of the characteristics of modern processors and provide fast batch “fixed point”¹ scalar multiplication for binary ECC.

Our Contribution. We propose a set of three tools

1. to optimize bitsliced binary field arithmetic, potentially supporting any architecture, by selecting the performance-optimal configuration of the underlying finite field layer for a given platform;
2. to implement an ECC layer for any given binary curve, building on the layer provided by the first tool, and providing constant-time fixed point scalar multiplications that performs very competitively with existing state-of-the-art results;
3. to integrate the generated ECC implementations in OpenSSL, overcoming the restriction of batch computation at the application level.

The combined output of these tools transparently provides constant-time and efficient implementations of bitsliced binary ECC for real-world applications built on top of OpenSSL. This challenges once again the notion that binary ECC are not well suited for software implementations, and at the same time overcomes the main drawback of similar techniques, proving it is also practical for real-world deployment.

Overview. In Section 2 we discuss the background for this work, recalling related works on top of which we build our contribution or that pursued similar goals. In Section 3 and Section 4 we discuss definitions, challenges, the design, and the analysis of our results related with the binary field and elliptic curve arithmetic layers. Section 5 describes the third and last of our contributions, integrating our implementations in OpenSSL to provide seamless support of constant-time and fast bitsliced binary ECC to real-world applications. Finally, we conclude in Section 6, with a discussion of the limits of our current contribution and future work directions for this research.

2 Background

2.1 Bitslicing

SIMD is a data parallelism technique that facilitates parallel computation on multiple values. For example, processors featuring AVX2 contain 256-bit registers

¹ In this paper we refer to scalar multiplication by the conventional generator point for a given curve as “*fixed point*” scalar multiplication. In real world applications, this is the fundamental operation for ECDH and ECDSA key generation, and for ECDSA signature generation. This operation is opposed to “*generic point*” scalar multiplication, intended as a scalar multiplication by any other point on the curve: the latter is used in ECDH key derivation and ECDSA signature verification.

`yymm0` through `yymm15`. Viewing these as four 64-bit *lanes*, the instruction `vpaddq %ymm0, %ymm1, %ymm2` takes $\text{ymm0} = (a_0, a_1, a_2, a_3)$ and $\text{ymm1} = (b_0, b_1, b_2, b_3)$ then produces their integer vector sum $\text{ymm2} = (a_0 + b_0, \dots, a_3 + b_3)$ where all sums are modulo 2^{64} . There are several ways to split the register; e.g. as 8-bit, 16-bit, etc. lanes, each requiring dedicated microarchitecture support and distinct instructions for the register-size variants (e.g. add, subtract, multiply, etc.). Put briefly, *bitslicing* takes this to the extreme and views any w -bit register as w 1-bit lanes. This lightens the microarchitecture requirements, as 1-bit addition (or subtraction) is simply bitwise-XOR; 1-bit multiplication is bitwise-AND. Bitwise operations are a fundamental feature in ISAs, and are in fact independent of explicit SIMD support: any generic, non-SIMD w -bit architecture natively supports 1-bit lane SIMD, i.e. bitslicing.

Deployments. Outside of primitives that integrate bitslicing into the design such as Serpent, SHA-3, and Ascon, we are aware of two large-scale deployments of bitsliced software: one defensive and the other offensive. Käsper and Schwabe [18] provide a bitsliced implementation for AES, mainlined by OpenSSL in 2011. In the pre-AES-NI era and with only SSE2 as a prerequisite, this was groundbreaking work that exceeded the performance of traditional table-based AES software, yet additionally provided timing attack resistance. *John the Ripper*² is a security audit tool that bitslices DES to batch password hashing. The main application is to hashes utilizing the `crypt` portion of the standard library.

Obstacles. The ability to batch public key operations is the largest restriction for bitsliced software. Real-world APIs (e.g. OpenSSL) do not support such a seemingly narrow use case, as e.g. in ECC the most common operations are single or double scalar multiplications, not w in parallel. Even research-oriented APIs such as SUPERCOP do not have this feature. And for key agreement on a typical single threaded application, there is no clear way how to utilize such an implementation. On the engineering side, it is very tempting to directly leverage academic results on minimizing gate count for bitsliced software, since each gate will map to an instruction. However, this is only part of the story since register and memory pressure impose constraints, as well as instruction level parallelism, scheduling, and binary size. Indeed, both [8, 34] note the latter disconnect. Binary finite field multipliers with low gate count [5] do not directly map to efficient bitsliced multipliers, since arithmetic instruction count is only a small part of overall performance on a platform.

2.2 OpenSSL and ENGINES

As already mentioned, arguably the main drawback of batch operations for cryptographic implementations is that they are generally seen as not practical: this comes in part by the lack of support for batch public key operations in mainstream cryptographic libraries. Among many others, one example supporting this argument is BBE251 by Bernstein [5]: despite the high performance, we are

² <https://www.openwall.com/john/>

not aware of any deployments or standardization efforts supporting this curve in the past decade. We hypothesize this is because it seemingly does not meet the characteristics for mainstream cryptography software.

To counter this argument and to evaluate our work in a real-world scenario, we decided to target OpenSSL, an open source project consisting of a general-purpose cryptographic library, an SSL/TLS library and toolkit, and a collection of command line tools to generate and handle cryptographic keys and execute cryptographic operations. The project is arguably ubiquitous, providing the cryptographic backend and TLS stack of a considerable portion of web servers, network appliances, client softwares, and IoT devices. Thanks to the wide range of supported platforms and more than twenty years of history, it has become a de facto standard for Internet Security.

In the literature, a common pattern to integrate alternative implementations in OpenSSL consists in forking the upstream project to apply the required patches. This then requires maintaining the fork to include, alongside the research-driven changes, patches from upstream to fix vulnerabilities, bugs, or provide new features. As an example of this methodology, the Open Quantum Safe project [30] maintains a fork³ of OpenSSL to evaluate candidates of the NIST Post Quantum Project. The history of the repository shows the level of effort required to maintain a fork of OpenSSL up to date with both research work and upstream releases.

Considering how demanding this approach can be, it is not surprising that most of the academic results often prefer to evaluate their results with ad-hoc software or toolkits like SUPERCOP⁴, which generally operate in isolation and are not necessarily representative of the performance or features of the applications of the research work in real-world use cases.

As an alternative to these two approaches, we build on top of the framework proposed by Tuveri and Brumley [33], instantiated in `libsuola`⁵. This framework allows to provide alternative implementations of cryptosystems to OpenSSL applications, by using `ENGINES`: OpenSSL objects that act as “containers for implementations of cryptographic algorithms”. Originally introduced to support hardware cryptographic accelerators, the same construct can be used to provide alternative software implementations. It offers a mechanism to configure a whole system or individual applications to load `ENGINES` at runtime, transparently providing their functionality to existing applications, without recompiling them.

While we defer to Section 5 for a description of the `ENGINE` instantiated as part of our contribution, we further motivate here our choice remarking that this approach, on top of lowering maintenance costs, allows to reuse existing applications with no effort, providing multiple and diverse ways to validate the correctness and interoperability of our implementation. This also allows us to

³ <https://github.com/open-quantum-safe/openssl>

⁴ <https://bench.cr.yp.to/supercop.html>

⁵ <https://github.com/romen/libsuola>

use existing projects to benchmark and evaluate our contribution in comparison with state-of-the-art real-world implementations.

3 Binary Field Arithmetic

A binary extension field is a finite field of the form \mathbb{F}_{2^m} , where m is an integer called the dimension of the field and $m \geq 2$. Elements of the field \mathbb{F}_{2^m} can be expressed as polynomials of degree less than m with coefficients in \mathbb{F}_2 , which have an underlying set of $\{0, 1\}$ and in which addition corresponds to binary XOR and multiplication corresponds to binary AND. Any two finite fields with the same number of elements are isomorphic to each other, but calculations in a particular finite field are performed modulo an irreducible polynomial P of degree m , and different choices of P produce different results. Since the underlying set of \mathbb{F}_2 is $\{0, 1\}$, elements of \mathbb{F}_{2^m} can also be represented as binary strings; for example, with $m = 8$, the element $x^6 + x^3 + x + 1$ can be written 01001011₂.

The simplest method of multiplying elements of \mathbb{F}_{2^m} is to multiply them as polynomials using schoolbook multiplication and then reduce the result modulo the field polynomial P by polynomial long division. Bernstein [5, Sect. 2] collects many asymptotic improvements on $M(n)$, the number of bit operations required to multiply two n -bit polynomials, over this method: $M(n) \leq \Theta(n^{\lg 3})$ due to Karatsuba, $M(n) \leq n2^{\Theta(\sqrt{\lg n})}$ due to Toom, and $M(n) \leq \Theta(n \lg n \lg \lg n)$ due to Schönhage and Strassen. Bernstein also establishes tighter explicit upper bounds on $M(n)$ for $n \in \{128, 163, 193, 194, 512\}$, and provides⁶ straight-line code for cases from $n = 1$ to $n = 1000$, and verified upper bounds on the number of bit operations required in each case.

3.1 Splitting Strategies

For small n , the straight-line code can be very efficient, but for large n , it becomes inefficient, partly because the compiled code becomes too large to fit in the cache. Additionally, the algorithm that uses the fewest bit operations will not necessarily take the fewest cycles to run when implemented, because in bitsliced batch computations a nontrivial number of cycles are spent performing load and store instructions, which is not accounted for in the bit operation count. There are several conventional concerns about the correlation of bit operation count and software performance, and while bitslicing relieves some of these concerns [5, Sec. 1], the overhead incurred by loads and stores remains relevant. This is a gap in the existing literature, which mostly reports results in terms of bit operations [11, 10, 16].

Recursive algorithms for polynomial multiplication, such as Karatsuba and Toom, have better asymptotic performance than straight-line multiplication, but they incur more overhead, so for sufficiently small inputs, straight-line multiplication is faster in practice. Due to these considerations, the fastest way (in terms

⁶ <https://binary.cr.yp.to/m.html>

of cycle count) to batch multiply polynomials of cryptographic sizes in \mathbb{F}_{2^m} is usually to begin with recursive splits, and then switch to straight-line multiplication when the subproblem size becomes small enough for this trade-off to occur. The exact size at which this threshold is located may depend on both the architecture and the field dimension.

For larger subproblems, there are many different recursive algorithms for polynomial multiplication. Of the recursive multiplication strategies described in [5], we use two: the one called “five-way recursion”, which corresponds to the WAY3 macros in our implementation, and the one called “two-level seven-way recursion”, which corresponds to the WAY4 macros. (There is also “three-way recursion”, corresponding to WAY2 macros, but we omit them; since WAY4 splits the current problem into four subproblems of roughly equal size, and WAY2 splits it into two subproblems of roughly equal size, we expect that WAY4 is a more efficient version of back-to-back WAY2 splits.) For simplicity, and in keeping with the names of the macros, in this paper we refer to the WAY3 macros as “three-way recursion” and to the WAY4 macros as “four-way recursion”. Different subproblem sizes may have different optimal choices of recursion strategy (and again, these can be architecture-dependent), so the complete collection of recursive multiplication steps taken before dropping down to straight-line multiplication may be both architecture- and dimension-dependent.

For a given field size, on a given architecture, a strategy to choose whether to use a recursive algorithm or switch to a straight-line multiplication at each intermediate step needs to be created. Said strategy aims to minimize the total number of cycles required to multiply a batch of w polynomials of the given degree. We refer to this result as the optimal *splitting strategy* for that size. The strategy is generated for the library and necessary straight-line multiplication files are included.

The functions for straight-line multiplication are called `gf2_mul_M` and the functions for recursive multiplication are called `karatmultM`, where M is the size of the input. Reading Figure 1 from the bottom line up, this code splits 251 four ways with a WAY4 macro (specifically WAY43 because $251 \equiv 3 \pmod{4}$) and performs recursive multiplication on subproblems of size 62 and 63; 63 is split four ways with a remainder of 3 and 62 is split four ways with a remainder of 2; both 62 and 63 are split into subproblems of size 15 and 16, which are handled with straight-line code.

```
/* (43K251, 43K63, 42K62, G16, G15) */
WAY42(62, gf2_mul_15, gf2_mul_16)
WAY43(63, gf2_mul_15, gf2_mul_16)
WAY43(251, karatmult62, karatmult63)
```

Fig. 1. Optimal splitting strategy on Skylake for $m = 251$.

When reporting benchmarking results, we display the splitting strategy in a concise format: a list of multipliers in descending order of subproblem size, with recursive multipliers represented by the numbers after the **WAY** macro + **K** + the input size, and straight-line multipliers represented by **G** + the input size.

Architectures. The purpose of the benchmarking tool we developed is to experimentally determine the best splitting strategy for a particular field size running on a particular architecture. The tooling currently supports AVX2, AVX-512, and NEON, but is easily extendable to other ISAs. The bulk of our code utilizes macros for C compiler intrinsics to emit architecture-specific instructions, so adding an ISA consists mainly of internally defining these macros for the target architecture. We used the following environments for the benchmarking: *AVX-512*, a 2.1GHz Xeon Silver 4116 Skylake (24 cores, 48 threads across 2 CPUs) and 256GB RAM running 64-bit Ubuntu 16.04 Xenial (`clang-8`, $w = 512$); *AVX2*, a 3.2GHz i5-6500 Skylake (4 cores) and 16GB RAM running 64-bit Ubuntu 18.04 Bionic (`clang-8`, $w = 256$); *AVX2-AMD*, a 3.7GHz Ryzen 7 2700X (8 cores, 16 threads) and 16GB RAM running 64-bit Ubuntu 18.04 Bionic (`clang-8`, $w = 256$); *NEON*, a Raspberry Pi 3 Model B+, 1.4GHz Broadcom BCM2837B0 ARMv8 Cortex-A53 (4 cores) and 1GB RAM running 64-bit Ubuntu 18.04 Bionic (`clang-7`, $w = 128$).

3.2 Benchmarking

The benchmarking for the new software was done by recursively generating different splitting strategies for the multiplication of the polynomials. The dimension of the original field is recursively split three-ways and four-ways until the limits for straight-line multiplication are reached. While within the limits, strategies are generated for both recursive and straight-line multiplication, because we found that using straight-line multiplication was not always the optimal solution even when they were available. Thus we also generate many strategies that still use the recursive split while within the limits of straight-line multiplication. While Bernstein [5] has straight-line multiplications defined in a larger range, we decided to limit it between polynomials of degree $[5, \dots, 99]$ for the scope of this paper.

We divided the generated splitting strategies into three categories, two of which were eliminated from this paper. The two eliminated categories consisted of *mixed multiplication* and *non-strict recursion threshold*. The nature of these categories and the reasons for elimination is discussed below.

The *mixed multiplication* category includes all the strategies where at least one subproblem of the recursive call is not handled like the others. An example of this would be `WAY43(251, karatmult62, gf2_mul_63)`, where one subproblem is handled with a recursive call, and another with straight-line multiplication. The benchmarking tool still supports this option, but our preliminary testing showed no benefits for allowing mixed multiplication. The number of possible strategies is also vastly larger (6 vs. 11 with degree 63, 14 vs. 62 with degree 127 and 193 vs. 4546 with degree 251), rendering the tool prohibitively slow with

higher values. Though there may be some edge cases where using a recursive call for the higher value and straight-line for the lower value subproblem would yield a more optimal result, we found none.

The *non-strict recursion threshold* category includes all the strategies where the greatest value of straight-line multiplication is greater or equal to the least value of a recursive multiplication. An example of this would be the strategy (43K251, 30K63, 32K62, G23, 42K22, 41K21, G21, 40K20, G6, G5), where we see both a G23 which has a greater value than K22, and K21 which is equal in value to G21. Our assumption is that once we cross the threshold on straight-line multiplication, using recursive multiplication will be slower. As is the case with mixed multiplication, limiting the search space by excluding this category considerably increases the efficiency of the benchmarking tool. We believe that this elimination does not significantly reduce the chances of finding the optimal strategy. If straight-line and recursive multiplication of the same subproblem size, such as G21 and K21, is wanted inside one strategy, the current version of the benchmarking software needs to be modified.

Unlike the mixed multiplication strategies, the strategies with non-strict recursion threshold are only eliminated after all the strategies have been generated. The final search space includes all the strategies that did not meet the criteria for elimination in the previous two steps.

After the paths have been generated, the benchmarking tool takes one strategy at a time, creates a configuration header file for the binary field arithmetic C program, and compiles and runs the test harness, which outputs the number of processor cycles it takes for a batch of polynomials to be multiplied. When all the strategies for that field size have been tested, the program outputs a file containing the strategies in ascending order of cycles, as well as the configuration file for the best found strategy for a given platform. Our ECC layer (Section 4) uses this configuration at build time to produce the most efficient solution.

Figure 2 shows the results of the best strategy in benchmarking on three different processor architectures, and Table 1 selective data points on four different processor architectures. NEON has a register width of 128 bits, AVX2 has a register width of 256 bits, and AVX-512 has a register width of 512 bits; these are the denominators by which the total cycle counts are scaled.

In theory, AVX-512 should perform twice as fast as AVX2 in terms of scaled cycle counts, because AVX-512 processes twice as many elements in one batch. Performing linear regression on the data sets in Figure 2 with `gnuplot` indicated that AVX-512 is faster than AVX2 by a factor of approximately 2.17 in practice.

We disabled Simultaneous multithreading (SMT) for all experiments in this paper. While we used multiple cores during benchmarking to find the best splitting strategy, the results in Table 1 (and later in Table 2) were ran on a single core.

3.3 Related Work

The cycle counts in this subsection are reported for $m = 251$ unless otherwise noted, since it is a common field size in the literature to approach the 128-

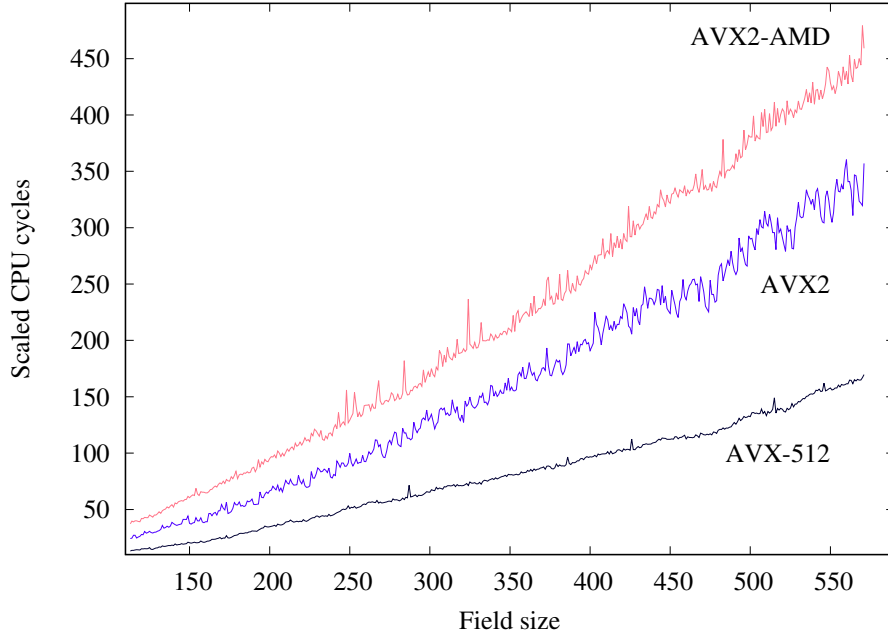


Fig. 2. Binary field multiplication performance in scaled CPU cycles.

Table 1. Selective binary field multiplication performance in scaled CPU cycles.

m	113	131	163	191	193	233	239	251	283	359	409	431	571
AVX-512	13	16	22	30	31	44	44	51	59	82	99	104	170
AVX2	24	30	43	54	55	84	77	91	121	166	216	214	351
AVX2-AMD	37	47	67	83	89	114	116	132	153	224	274	300	459
NEON	228	290	425	523	534	708	724	805	928	1340	1659	1819	2953

bit security level and therefore provides the best basis for comparison. We first note that the comparison here (and later in Section 4.5) to previous work is not without caveats, due to different computation models (i.e. parallel vs. serial) and ISA availability over time.

Aranha et al. [4] benchmarked several field operations on three different Intel platforms (Core 2 65nm, Core 2 45nm, and Core i7 45nm) Their best multiplication result was 323 cycles, achieved on the Core i7 platform with López-Dahab multiplication.

Taverne et al. [31, 32] benchmarked López-Dahab multiplication on an Intel Westmere Core i5 32nm processor, reporting 338 cycles for code compiled with GCC and 429 cycles for code compiled with ICC (the Intel C++ Compiler), and achieved a speedup to 161 cycles (GCC) and 159 cycles (ICC) by performing Karatsuba multiplication with the new carry-less multiplication instruction.

Câmara et al. [9] report timings for an ARM Cortex-A8, Cortex-A9, and Cortex-A15 processor, with code compiled with GCC. López-Dahab multiplication takes 671 cycles on A8, 774 on A9, and 412 on A15; their contribution is the Karatsuba/NEON/VMULL multiplication algorithm, which takes 385 cycles on A8, 491 on A9, and 317 on A15.

Oliveira et al. [25] benchmarked Karatsuba multiplication with carry-less multiplication on Intel Sandy Bridge for field size $m = 254$ and achieved 94 cycles with GCC (and report similar results for ICC).

Seo et al. [27] achieve 57 cycles for $m = 251$ and 153 cycles for $m = 571$ on ARMv8 by using the 64-bit polynomial multiplication instruction PMULL instead of the 8-bit polynomial multiplication instruction VMULL used in [9].

4 Elliptic Curve Arithmetic

A non-supersingular elliptic curve E over the binary finite field \mathbb{F}_{2^m} is a set of points $(x, y) \in \mathbb{F}_{2^m}$ satisfying the short Weierstrass equation:

$$E(\mathbb{F}_{2^m}) : y^2 + xy = x^3 + ax^2 + b$$

where the parameters $a, b \in \mathbb{F}_{2^m}$ and the set of points in $E(\mathbb{F}_{2^m})$ is an additive Abelian group with identity element as the point at infinity \mathcal{O} i.e. $P + \mathcal{O} = \mathcal{O} + P = P$ and the inverse element at $-P = (x_1, x_1 + y_1)$. For the sake of simplicity, the remainder of this paper will refer to this curve form as *short curves*.

4.1 Scalar Multiplication

The elliptic curve point multiplication is seen as the most critical and compute-intensive task, therefore a lot of emphasis is given on improving the algorithms for efficiency and security. For a given ℓ -bit scalar $k \in \mathbb{Z}$ and point $P \in E$, the point multiplication can be formulated as:

$$kP = \sum_{i=0}^{\ell-1} k_i 2^i P$$

where k_i is the i_{th} bit of the scalar. This naive method, a.k.a. binary method, scans the bits of the scalar one at a time, performing double operations for each $\ell - 1$ bits in addition to the add operation where $k_i \neq 0$.

An alternate approach was proposed by Montgomery [24]: where the same number of double-and-add operations are repeated in each step. Although—in terms of computational time—it seems more expensive as compared to the naive method. However, the advantage is the resistance against side-channel analysis by removing the conditional branches depending on the weight of scalar bits.

For performing fast ladder multiplication, differential addition and doubling is applied, which computes $P_1 + P_2$ from P_1 , P_2 and $P_2 - P_1$ and similarly

$2P_1$ from P_1 for the doubling. The relation $P_2 - P_1 = P$ here is the invariant, i.e. the difference of these two points is known and constant throughout the ladder step. Short curves have a nice property, by applying suitable coordinate transformation, the knowledge of only x -coordinate is sufficient to perform the entire ladder step. This considerably improves the performance, since there is no need to evaluate the intermediate results of the y -coordinate at each ladder step.

An important consideration—for the performance optimization of EC double and add operations—is the representation of point coordinate system. The use of projective coordinates in most cases is preferred over the affine coordinates, due to the heavy inversion operations involved in the later. A number of coordinate systems, in the context of binary curves, have been studied such as lambda [25], Jacobian [12], and homogeneous projective [1]. Among them, López and Dahab [21] is a popular choice in binary elliptic curves, with various performance improvements proposed [2, 20, 7]. For a projective form of the short curves equation defined as:

$$E(\mathbb{F}_{2^m}) : Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4$$

the LD-projective point $(X_1 : Y_1 : Z_1)$ is equivalent to the affine point $(X_1/Z_1 : Y_1/Z_1^2)$, where the inverse is $(X_1 : X_1Z_1 + Y_1 : Z_1)$. It is further assumed that $Z_1 \neq 0$ and \mathcal{O} is not among the points.

4.2 Differential Montgomery Ladder

For short curves, Bernstein and Lange⁷ provide a very efficient differential addition and doubling (*ladder step*) formulas, originally derived from [29]. The representation, known as XZ , is a variant of the original LD coordinates. For each scalar bit k_i —assuming the invariant $P_2 - P_1 = P$ is known—the fast differential addition and doubling takes 5 multiplications, 4 squares and 1 multiplication by the constant \sqrt{b} . As mentioned by [14], this is the best case bound achieved for Montgomery ladder also on other forms of curves, such as Hessian [15], Huff [13], and Edwards [7], which means that choice of curve form is mostly irrelevant in terms of the computational cost. It is still possible to achieve slight performance gains over this bound by leveraging efficient forms of subfield curve constants [3].

Given $P_1 \neq \pm P_2$, to compute the differential point addition $P_3 = P_1 + P_2 = (X_3 : Y_3 : Z_3)$ where $P_1 = (X_1 : Y_1 : Z_1)$, $P_2 = (X_2 : Y_2 : Z_2)$ and, point doubling $P_4 = 2P_2 = (X_4 : Y_4 : Z_4)$, the formulas are defined as follows.

$$\begin{aligned} A &= X_1Z_2, B = X_2Z_1, C = X_1^2, D = Z_1^2 \\ Z_3 &= (A + B)^2 \\ X_3 &= XZ_3 + AB \\ X_4 &= (C + \sqrt{b}D) \\ Z_4 &= CD \end{aligned}$$

⁷ <https://hyperelliptic.org/EFD/g12o/auto-shortw-xz.html>

As previously mentioned, the Montgomery ladder differential addition and double method does not take into account the y -coordinate during the computation, however to get back to the affine coordinates we need to recover the y -coordinate. López and Dahab [22] presented a formula to retrieve the y -coordinate, which enabled the complete point multiplication using the Montgomery ladder method, i.e. also compute the resulting affine point. In our case we applied a further optimization to the original formula by setting $Z = 1$ since the invariant $P_2 - P_1 = P = (X : Y : Z)$ is fixed, the resulting simplified version is formulated as:

$$x_1 = \frac{xZ_2X_1}{xZ_2Z_1}$$

$$y_1 = y + \frac{(X_2 + xZ_2)((X_1 + xZ_1)(X_2 + xZ_2) + Z_1Z_2x^2 + Z_1Z_2y)}{Z_1Z_2x}$$

The cost of recovering y is 10 multiplications and 1 inversion, which is small, considering that it is performed just once at the end of the ladder.

4.3 Linear Maps

As a further optimization, we also considered replacing the multiplication by the curve constants, and x -coordinate of the generator point during the ladder step by a linear map. For a finite field \mathbb{F}_{2^m} represented as m -dimensional vector space $(\mathbb{F}_2)^m$, the linear map $T : \mathbb{F}_{2^m} \mapsto \mathbb{F}_{2^m}$ for multiplication by a constant b is an $(m \times m)$ matrix B with entries in \mathbb{F}_2 . The elements b_i in B can be pre-computed: for a basis α of degree m , $b_i = \alpha^i b$ for $0 \leq i < m$, where b is the element of vector space $(\mathbb{F}_2)^m$. The multiplication by an element $a \in \mathbb{F}_{2^m}$ can then be replaced by $b \cdot a = Ba$, where each bit in the product can be written as $\sum_{i: b_{i,j}=1} a_i$.

This essentially means that if there are l non-zero bits for each row of the matrix B , in total we need $(l-1)m$ XOR gates for the entire matrix. This is not the optimal number of gates, since common sub-expression elimination is not accounted for. For this purpose we tried optimization as suggested by Bernstein [6] which produces approximately $ml/(\log l - \log \log(l))$ XORs, in addition to m number of copies of the intermediate results.

For hardware implementations, this straight-line optimization makes sense. However, in the software case this also increases both register pressure and, more importantly, binary size. For the above ladder step formula, we applied the tooling from [6] to replace multiplications by both the curve constant and fixed generator x -coordinate by the straight-line code. The resulting binary far exceeded the L1 cache size, and benchmark results showed it takes away any performance gains when compared to generic finite field multiplication in the ladder step.

4.4 Implementation

Our bitsliced ECC layer builds on our finite field layer from Section 3. As a library, it exposes a single function `EC2_batch_keygen` that takes four arguments:

(1) `EC2_CURVE` pointer to an opaque object provided by the library to represent a specific curve; (2) `unsigned char` pointer `k`, where it is assumed the caller has filled this input with enough randomness for w scalars, and from which the caller will retrieve the scalars as an output; (3) `unsigned char` pointer `x` storing the resulting x -coordinate of the scalar multiplication by G ; and similarly (4) `unsigned char` pointer `y` for the y -coordinate. Geared to ease real-world deployment discussed further in Section 5, since our application is *only* to fixed (generator) point scalar multiplication, `EC2_batch_keygen` internally makes a number of optimizations.

We assume the scalars are provided in bitsliced form, simply viewing the first w bits at `k` as bit 0 of the scalars, and so on for the rest of the bits. This imposes no practical requirement on the caller—they are just random bytes, only viewed differently.

As previously discussed, for the range of standardized curves under consideration in this work, the linear maps for constant multiplications are not efficient when bitslicing, hence we implement them as generic finite field multiplications with fixed pre-bitsliced form stored in the binary. The *exception* to this case is `curve2251`, which has small and sparse curve coefficients. Here we implemented a dedicated ladder step using a slightly different formula involving the curve coefficient b and replace the multiplication with a straight-line linear map. Since this curve is not fixed by a standard, we found the lexicographically smallest x such that the resulting generator satisfies $\text{ord}(G) = n$, the large prime subgroup order. We then replaced this multiplication (by x -coordinate `e`) in the ladder step with a straight-line linear map. Of course we could do the same for standardized curves, but this would violate interoperability.

After scalar multiplication, `EC2_batch_keygen` recovers the y -coordinate, since key generation is our motivating use case and not simply key agreement where the x -coordinate is sufficient. Our finite field layer uses Itoh-Tsujii [17] in the inversion step, where the addition chain for exponentiation is efficient and fixed based on the finite field degree. Our finite field layer also supplies efficient finite field squaring with a straight-line linear map.

Finally, `EC2_batch_keygen` converts the outputs for `k`, `x`, and `y` from bitsliced form to canonical form for consumption by linking applications. Note for our use case of key generation, there are no exceptions in the ladder step, the y recovery, or scalar corner cases. We completely control all scalars and points.

4.5 Benchmarking

Our tooling for the ECC layer generates and exposes an `EC2_CURVE` object for each fixed curve. We restrict to short curves with (1) no efficient endomorphism and (2) field degrees that are prime. Table 2 reports the performance across our four target architectures. Each curve was benchmarked using the best splitting strategy for its respective underlying field, as measured by the benchmarking process described in Section 3. Similar to Section 3, the reported cycle counts are scaled, dividing by w .

Table 2. ECC performance on four architectures in scaled CPU cycles.

Curve	AVX-512	AVX2	AVX2-AMD	NEON
sect113r1	9547	18074	27470	153944
sect113r2	9540	17962	27487	153948
sect131r1	13684	26821	40478	227765
sect131r2	13639	26856	40466	228168
sect163r1	22849	45231	70046	427274
sect163r2	23175	46826	70413	448744
c2pnb163v1	23005	45017	74888	435651
c2pnb163v2	22881	45490	74413	426473
c2pnb163v3	22805	45380	74422	429631
c2tnb191v1	36094	66799	102005	632907
c2tnb191v2	36262	64963	101235	617958
c2tnb191v3	35680	64454	101325	629464
sect193r1	37899	67325	109376	639270
sect193r2	37936	67730	109434	640406
sect233r1	65491	125804	167761	1013458
c2tnb239v1	67144	119490	175914	1079930
c2tnb239v2	67105	117703	174388	1085560
c2tnb239v3	67181	120304	175676	1063116
curve2251	57756	106391	146031	870376
sect283r1	105304	218130	272544	1595423
c2tnb359v1	186680	362665	504219	2961857
sect409r1	260619	546690	697021	4229741
c2tnb431r1	283319	567608	780886	4812995
sect571r1	627668	1303759	1629335	10676160

For the sake of discussion, our results show it is more efficient to use canonical representations of the curve constants, including curve coefficients and x -coordinate of the generator point. This saves in terms of computational cost during the differential addition step, allowing `curve2251` to significantly outperform curves at comparable field sizes.

Finally, we briefly compare with selective results from the literature. Bernstein [5] reports 314K (scaled, SSE2, $w = 128$) Core 2 cycles for the binary Edwards curve `BBE251`. Aranha et al. [4] report 537K, 793K, and 4.4M Core i7 cycles for curves `curve2251`, B-283, and B-571. Taverne et al. [32] report 225K Sandy Bridge cycles for `curve2251` in constant time, 100K cycles for B-233 in non constant time, and 349K cycles for B-409 in non constant time. Oliveira et al. [25] report 114K Sandy Bridge cycles for a 254-bit curve in constant time, yet with a non-standard composite extension. Câmara et al. [9] report 511K, 866K, and 4.2M ARM Cortex-A15 cycles for constant time `curve2251`, B-283, and B-571. Oliveira et al. [26] report 46K Skylake cycles for a 254-bit curve in constant time, yet again with a non-standard composite extension.

Generally, even without focusing on a single curve yet imposing the batch computation requirement, the results in Table 2 compare very favorably with the existing literature. Considering it is automated tooling that generates the finite

field layer, and the ECC layer utilizes a stock ladder with no fast endomorphisms or precomputation, this demonstrates the tooling has wide applicability and can provide a strong baseline for performance comparison.

5 ENGINE implementation

As mentioned in Section 2, we integrate our bitsliced implementation in OpenSSL through an ENGINE, dubbed `libbecc`, modeled after `libsuola` [33].

We defer to [33] for a detailed description of the ENGINE framework and how it integrates with the OpenSSL architecture, while in this section we provide an overview of the design of `libbecc` in comparison to `libsuola`.

Our ENGINE provides implementations for most of the named⁸ binary curves defined in OpenSSL and adds dedicated support to `curve2251`; this is achieved building on top of the work described in Sections 3 and 4, which provide the actual batch implementation for elliptic curve and binary field operations. The `libbecc` code mainly provides an interface to query the underlying layers and to dispatch to the relevant codepath when, through the OpenSSL library, an application requests a cryptographic operation that requires computation over a supported curve and field.

5.1 Providers

The other fundamental function of `libbecc` is to implement the logic to maintain a state for each performed batch operation, so that following requests can be served from the precomputed results rather than issuing a new batch operation.

We achieve support for batch ECC operations using the ENGINE API to register `libbecc` as the default `EC_KEY_METHOD`: by doing so our ENGINE is activated for any operation involving an `EC_KEY` object, including ECDH and ECDSA key generation, ECDH shared secret derivation and ECDSA signature generation and verification. `libbecc` retains a reference to the default OpenSSL `EC_KEY_METHOD`, which is used to bypass our ENGINE for unsupported curves or for operations such as ECDH derivation or ECDSA verification: these operations are not supported by our bitsliced code, limited to “fixed point” scalar multiplications, i.e., limited to scalar multiplications by the conventional generator point for a given curve.

Following `libsuola` approach and terminology, `libbecc` supports the notion of multiple providers to interface with the OpenSSL API, by providing a minimum set of functions to:

⁸ The term “named” here is used in contrast with curves described by arbitrary parameters: usage in real-world applications is dominated by curves that have been assigned code points as part of standards, delivering both security assurances on the cryptographic features and security evaluation of the group defined by the specified set of parameters, and saving the users from the need of performing expensive validation of the group parameters during curve negotiation.

- match an EC object with any of the supported curves, returning either *unsupported* or a provider specific integer identifier for the specific curve;
- generate one key for a given curve identifier, returning a secret random scalar and the corresponding public point;
- perform the setup step of ECDSA signature generation for a curve identifier, returning the modular inverse of a secret nonce scalar and the corresponding r component of a (r, s) ECDSA signature.

Providers can then internally differ on the way the batch logic is implemented to support a bitsliced scalar multiplication.

Provider: serial. Specifically, we instantiate one provider, dubbed `serial`, that stores the internal state and buffers for key generation and signature setup with a thread local model. In this model, each thread of an OpenSSL application loading our `ENGINE` stores its own local state and buffers, and the batch results are not shared across separate threads.

In our design, we opted to perform all the operations required by a specific cryptosystem operation during the batch computation, including the conversion of raw binary buffers in OpenSSL `BIGNUM` objects, and, during the batch computation for `sign_setup`, also the batch inversion of the original nonces. To improve performance, we implemented the batch inversion using the so-called Montgomery trick [24, Sect. 10.3.1] which allows to compute simultaneously the inverses of n elements at the cost of 1 inversion and $(3n - 1)$ multiplications [28, Sect. 3.1].

The design of the `serial` provider is the most straightforward strategy to implement batch operations in OpenSSL. It avoids synchronization issues and allows us to evaluate the performance of our implementation compared to the baseline upstream implementation with classic benchmarking tools, as they usually consist of a serial loop of repeated operations.

On the other hand, we acknowledge that this is not the optimal implementation for real-world high-load multi-threaded or multi-process applications, which would actually benefit the most from bitsliced operations, saving memory resources and minimizing the number of batch operations to run, if a more clever logic to share the results of batch operations were implemented. In particular, although of limited academic value for this paper, it would be interesting to add a provider supporting inter-process communication, to have a separate system-wide *singleton* service in charge of running the batch operations and storing the results, while applications using `libbecc` would simply request a fresh result from the service. Such design would minimize memory consumption and the number of batch operations across the whole system, and it would also provide a stronger security model to protect the state and buffers in memory, as they would be stored in a separate process space. Leveraging the access control capabilities of the underlying operating system, this would be inaccessible from a compromised application.

5.2 Benchmarking

To evaluate the actual practical impact of the presented improvements, we instantiated a benchmarking application built on top of OpenSSL 1.1.1. Table 3 reports the average number of CPU clock cycles to compute a single ECDH/ECDSA key generation and ECDSA signature generation. We compare the results recorded against the default implementation with a run of the same application after loading `libbecc` at runtime; due to space constraints we limit this analysis to the AVX2 platform. For comparison, we also include measurements relative to operations on top of popular prime curves, namely ECDH/ECDSA over `secp256r1` (a.k.a. NIST P-256), and EDDSA over `ED25519` and `ED448`.

The benchmarking application consecutively runs 2^{16} operations for each curve, recording the number of elapsed CPU cycles after each operation; Table 3 reports the average of such measurements. It should be noted that, for the default implementation, each measure for a given operation on a given curve is relatively close to the average reported on the table; for the `libbecc` implementation instead, due to the nature of batch operations, we record execution time spikes when a new batch is computed followed by a relatively low (between nine and sixteen thousands of CPU cycles, depending on the field size of the curve) plateau for each following operation until the batch is consumed.

6 Conclusion

In this paper, we developed a tool to optimize bitsliced binary field arithmetic that can potentially support any architecture. Guided by benchmarking statistics, at the finite field layer the tool tries different polynomial splitting strategies to arrive at the performance-optimal configuration on a given platform. Building on this layer, we developed a second tool that implements an ECC layer for a given binary curve, and performs very competitively; e.g. 58K AVX-512 (scaled) cycles for constant-time fixed point scalar multiplication on `curve2251`. Building on both these results, our last layer links OpenSSL with the output, overcoming the restriction of batch computation at the application level. Our approach in `libbecc` seamlessly couples applications with the batch computation results, facilitating real-world deployment of bitsliced public key cryptography software.

Future work. Our ECC layer is specific to short curves; a natural direction is to extend with support for other binary curve forms, even using the birational equivalence with short curves where applicable to maintain compatibility with existing (legacy, X9.62) standards exposed by OpenSSL through `libbecc`. Lastly, the architecture of `libbecc` has several applications outside binary ECC. Exploring similar functionality for traditional SIMD instead of bitslicing is another research direction, providing batch computation for curves over prime fields. Such an implementation would have much lower batch sizes, but potentially far greater performance since it relaxes register and memory pressure.

Acknowledgments. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and

Table 3. OpenSSL performance on AVX2: average CPU cycles for operations on selected curves.

Curve	Average CPU cycles per operation			
	Key generation		Signature generation	
	default	libbecc	default	libbecc
sect113r1	309998	26758 (11.6x)	323163	25049 (12.9x)
sect113r2	309892	26810 (11.6x)	323586	25043 (12.9x)
sect131r1	515314	37617 (13.7x)	533484	35950 (14.8x)
sect131r2	519635	37293 (13.9x)	540221	35711 (15.1x)
c2pnb163v1	699094	54717 (12.8x)	718671	52785 (13.6x)
c2pnb163v2	690930	54603 (12.7x)	710865	52653 (13.5x)
c2pnb163v3	700345	54432 (12.9x)	722117	52709 (13.7x)
sect163r1	690258	53996 (12.8x)	719847	52548 (13.7x)
sect163r2	697992	54875 (12.7x)	725706	52635 (13.8x)
c2tnb191v1	673839	74407 (9.1x)	694368	72989 (9.5x)
c2tnb191v2	668479	74308 (9.0x)	695895	72811 (9.6x)
c2tnb191v3	669240	73836 (9.1x)	697687	73037 (9.6x)
sect193r1	762628	77378 (9.9x)	803603	76853 (10.5x)
sect193r2	758937	77109 (9.8x)	800200	76908 (10.4x)
sect233r1	940852	133436 (7.1x)	985741	133941 (7.4x)
c2tnb239v1	966659	127149 (7.6x)	1008116	126843 (7.9x)
c2tnb239v2	960048	126222 (7.6x)	1004913	126053 (8.0x)
c2tnb239v3	961976	125478 (7.7x)	1008270	125681 (8.0x)
curve2251	1139439	118368 (9.6x)	1198634	118661 (10.1x)
ED25519	130295	130254 (1.0x)	131174	129597 (1.0x)
secp256r1	35805	36741 (1.0x)	69228	68921 (1.0x)
sect283r1	1631437	226075 (7.2x)	1700907	225973 (7.5x)
c2tnb359v1	2016295	377226 (5.3x)	2126677	374781 (5.7x)
sect409r1	2731705	552476 (4.9x)	2880190	551485 (5.2x)
c2tnb431r1	2968140	587026 (5.1x)	3125245	579913 (5.4x)
ED448	960595	957772 (1.0x)	969825	969300 (1.0x)
sect571r1	6283098	1359731 (4.6x)	6624862	1311432 (5.1x)

innovation programme (grant agreement No 804476). The second author was supported in part by the Tuula and Yrjö Neuvo Fund through the Industrial Research Fund at Tampere University of Technology.

References

1. Agnew, G.B., Mullin, R.C., Vanstone, S.A.: An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications* 11(5), 804–813 (1993), <https://doi.org/10.1109/49.223883>
2. Al-Daoud, E., Mahmud, R., Rushdan, M., Kiliçman, A.: A new addition formula for elliptic curves over $GF(2^n)$. *IEEE Trans. Computers* 51(8), 972–975 (2002), <https://doi.org/10.1109/TC.2002.1024743>
3. Aranha, D.F., Azarderakhsh, R., Karabina, K.: Efficient software implementation of laddering algorithms over binary elliptic curves. In: Ali, S.S., Danger, J., Eisen-

- barth, T. (eds.) Security, Privacy, and Applied Cryptography Engineering - 7th International Conference, SPACE 2017, Goa, India, December 13-17, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10662, pp. 74–92. Springer (2017), https://doi.org/10.1007/978-3-319-71501-8_5
4. Aranha, D.F., López, J., Hankerson, D.: Efficient software implementation of binary field arithmetic using vector instruction sets. In: Abdalla, M., Barreto, P.S.L.M. (eds.) Progress in Cryptology - LATINCRYPT 2010, First International Conference on Cryptology and Information Security in Latin America, Puebla, Mexico, August 8-11, 2010, Proceedings. Lecture Notes in Computer Science, vol. 6212, pp. 144–161. Springer (2010), https://doi.org/10.1007/978-3-642-14712-8_9
 5. Bernstein, D.J.: Batch binary Edwards. In: Halevi, S. (ed.) Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5677, pp. 317–336. Springer (2009), https://doi.org/10.1007/978-3-642-03356-8_19
 6. Bernstein, D.J.: Optimizing linear maps modulo 2. In: SPEED-CC: Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers, Workshop Record. pp. 3–18 (2009), <http://cr.yp.to/papers.html#linearmod2>
 7. Bernstein, D.J., Lange, T., Farashahi, R.R.: Binary Edwards curves. In: Oswald, E., Rohatgi, P. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5154, pp. 244–265. Springer (2008), https://doi.org/10.1007/978-3-540-85053-3_16
 8. Brumley, B.B., Page, D.: Bit-sliced binary normal basis multiplication. In: Antelo, E., Hough, D., Ienne, P. (eds.) 20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011. pp. 205–212. IEEE Computer Society (2011), <https://doi.org/10.1109/ARITH.2011.36>
 9. Câmara, D.F., Gouvêa, C.P.L., López, J., Dahab, R.: Fast software polynomial multiplication on ARM processors using the NEON engine. In: Cuzzocrea, A., Kittl, C., Simos, D.E., Weippl, E.R., Xu, L. (eds.) Security Engineering and Intelligence Informatics - CD-ARES 2013 Workshops: MoCrySEn and SeCIHD, Regensburg, Germany, September 2-6, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8128, pp. 137–154. Springer (2013), https://doi.org/10.1007/978-3-642-40588-4_10
 10. Cenk, M.: Karatsuba-like formulae and their associated techniques. J. Cryptographic Engineering 8(3), 259–269 (2018), <https://doi.org/10.1007/s13389-017-0155-8>
 11. Cenk, M., Hasan, M.A.: Some new results on binary polynomial multiplication. J. Cryptographic Engineering 5(4), 289–303 (2015), <https://doi.org/10.1007/s13389-015-0101-6>
 12. Chudnovsky, D., Chudnovsky, G.: Sequences of numbers generated by addition in formal groups and new primality and factorization tests. Advances in Applied Mathematics 7(4), 385–434 (1986), [http://dx.doi.org/10.1016/0196-8858\(86\)90023-0](http://dx.doi.org/10.1016/0196-8858(86)90023-0)
 13. Devigne, J., Joye, M.: Binary Huff curves. In: Kiayias, A. (ed.) Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6558, pp. 340–355. Springer (2011), https://doi.org/10.1007/978-3-642-19074-2_22

14. Farashahi, R.R., Hosseini, S.G.: Differential addition on binary elliptic curves. In: Duquesne, S., Petkova-Nikova, S. (eds.) Arithmetic of Finite Fields - 6th International Workshop, WAIFI 2016, Ghent, Belgium, July 13-15, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10064, pp. 21–35 (2016), https://doi.org/10.1007/978-3-319-55227-9_2
15. Farashahi, R.R., Joye, M.: Efficient arithmetic on Hessian curves. In: Nguyen, P.Q., Pointcheval, D. (eds.) Public Key Cryptography - PKC 2010, 13th International Conference on Practice and Theory in Public Key Cryptography, Paris, France, May 26-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6056, pp. 243–260. Springer (2010), https://doi.org/10.1007/978-3-642-13013-7_15
16. Find, M.G., Peralta, R.: Better circuits for binary polynomial multiplication. IEEE Trans. Computers 68(4), 624–630 (2019), <https://doi.org/10.1109/TC.2018.2874662>
17. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. Inform. and Comput. 78(3), 171–177 (1988), [https://doi.org/10.1016/0890-5401\(88\)90024-7](https://doi.org/10.1016/0890-5401(88)90024-7)
18. Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: Clavier, C., Gaj, K. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings. Lecture Notes in Computer Science, vol. 5747, pp. 1–17. Springer (2009), https://doi.org/10.1007/978-3-642-04138-9_1
19. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation 48(177), 203–209 (1987)
20. Lange, T.: A note on López-Dahab coordinates. IACR Cryptology ePrint Archive 2004, 323 (2004), <http://eprint.iacr.org/2004/323>
21. López, J., Dahab, R.: Improved algorithms for elliptic curve arithmetic in $GF(2^n)$. In: Tavares, S.E., Meijer, H. (eds.) Selected Areas in Cryptography '98, SAC'98, Kingston, Ontario, Canada, August 17-18, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1556, pp. 201–212. Springer (1998), https://doi.org/10.1007/3-540-48892-8_16
22. López, J., Dahab, R.: Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In: Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1717, pp. 316–327. Springer (1999), https://doi.org/10.1007/3-540-48059-5_27
23. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings. Lecture Notes in Computer Science, vol. 218, pp. 417–426. Springer (1985), https://doi.org/10.1007/3-540-39799-X_31
24. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comp. 48(177), 243–264 (1987), <https://doi.org/10.2307/2007888>
25. Oliveira, T., López, J., Aranha, D.F., Rodríguez-Henríquez, F.: Two is the fastest prime: lambda coordinates for binary elliptic curves. J. Cryptographic Engineering 4(1), 3–17 (2014), <https://doi.org/10.1007/s13389-013-0069-z>
26. Oliveira, T., López, J., Rodríguez-Henríquez, F.: The Montgomery ladder on binary elliptic curves. J. Cryptographic Engineering 8(3), 241–258 (2018), <https://doi.org/10.1007/s13389-017-0163-8>
27. Seo, H., Liu, Z., Nogami, Y., Choi, J., Kim, H.: Binary field multiplication on ARMv8. Security and Communication Networks 9(13), 2051–2058 (2016), <https://doi.org/10.1002/sec.1462>

28. Shacham, H., Boneh, D.: Improving SSL handshake performance via batching. In: Naccache, D. (ed.) Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2020, pp. 28-43. Springer (2001), https://doi.org/10.1007/3-540-45353-9_3
29. Stam, M.: On Montgomery-Like representations for elliptic curves over $\text{GF}(2^k)$. In: Desmedt, Y. (ed.) Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2567, pp. 240-253. Springer (2003), https://doi.org/10.1007/3-540-36288-6_18
30. Stebila, D., Mosca, M.: Post-quantum key exchange for the Internet and the Open Quantum Safe project. In: Avanzi, R., Heys, H.M. (eds.) Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10532, pp. 14-37. Springer (2016), https://doi.org/10.1007/978-3-319-69453-5_2
31. Taverne, J., Faz-Hernández, A., Aranha, D.F., Rodríguez-Henríquez, F., Hankerson, D., López, J.: Software implementation of binary elliptic curves: Impact of the carry-less multiplier on scalar multiplication. In: Preneel, B., Takagi, T. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6917, pp. 108-123. Springer (2011), https://doi.org/10.1007/978-3-642-23951-9_8
32. Taverne, J., Faz-Hernández, A., Aranha, D.F., Rodríguez-Henríquez, F., Hankerson, D., López, J.: Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction. *J. Cryptographic Engineering* 1(3), 187-199 (2011), <https://doi.org/10.1007/s13389-011-0017-8>
33. Tuveri, N., Brumley, B.B.: Start your ENGINES: dynamically loadable contemporary crypto. In: IEEE Secure Development Conference, SecDev 2019, McLean, VA, USA, September 25-27, 2019. IEEE Computer Society (2019), <https://eprint.iacr.org/2018/354>
34. Wiggers, T.: Energy-efficient ARM64 cluster with cryptanalytic applications: 80 cores that do not cost you an ARM and a leg. In: Progress in Cryptology - LATINCRYPT 2017 - 5th International Conference on Cryptology and Information Security in Latin America, La Habana, Cuba, September 20-22, 2017, Proceedings. LNCS, Springer (2017), <https://eprint.iacr.org/2018/888>

APPENDIX B: STRATEGY COUNTS

The following table contains the number of generated strategies for each field size for a straightline upper limit of 99 or 23, and information about how many of those strategies will be unsuitable for use because the compiled code produces incorrect results.

- The leftmost column n indicates the field size $GF(2^n)$.
- On other columns, “(SL k)” indicates that k is the largest subproblem size for which straightline code may be used (“SL” stands for “straightline limit”).
- “Total” is the total number of structurally valid strategies, under the constraints described in chapter 5.
- “Errors” is the number of valid strategies which fail the inversion test (i.e. we cannot use them because the code generated for them is incorrect). As discussed in chapter 6, these are exactly the strategies which use a three-way split for the subproblem of size 19. Since changing the straightline limit from 99 to 23 neither removes any of these strategies nor creates any additional ones, this column is only reported once. For ease of reading, zeroes are replaced with a dash.
- “Err. ratio” is the number of strategies with errors divided by the total number of strategies, formatted to 4 decimal places.

n	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
113	12	10	–	–	–
114	10	8	–	–	–
115	14	12	–	–	–
116	12	10	–	–	–
117	8	6	–	–	–
118	14	12	–	–	–
119	14	12	–	–	–
120	8	6	–	–	–
121	12	10	–	–	–
122	14	12	–	–	–
123	10	8	–	–	–
124	12	10	–	–	–

<i>n</i>	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
125	12	10	–	–	–
126	10	8	–	–	–
127	14	12	–	–	–
128	12	10	–	–	–
129	9	7	–	–	–
130	15	13	–	–	–
131	15	13	–	–	–
132	8	6	–	–	–
133	13	11	–	–	–
134	15	13	–	–	–
135	11	9	–	–	–
136	12	10	–	–	–
137	13	11	–	–	–
138	10	8	–	–	–
139	15	13	–	–	–
140	12	10	–	–	–
141	9	7	–	–	–
142	15	13	–	–	–
143	15	13	–	–	–
144	8	6	–	–	–
145	13	11	–	–	–
146	15	13	–	–	–
147	13	11	2	0.1538	0.1818
148	13	11	1	0.0769	0.0909
149	15	13	2	0.1333	0.1538
150	10	8	–	–	–
151	17	15	2	0.1176	0.1333
152	13	11	1	0.0769	0.0909
153	11	9	3	0.2727	0.3333
154	15	13	1	0.0667	0.0769
155	17	15	3	0.1765	0.2000
156	10	8	–	–	–
157	23	21	9	0.3913	0.4286
158	19	17	3	0.1579	0.1765
159	15	13	5	0.3333	0.3846
160	18	16	4	0.2222	0.2500
161	25	23	11	0.4400	0.4783
162	16	14	4	0.2500	0.2857

<i>n</i>	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
163	27	25	13	0.4815	0.5200
164	26	24	12	0.4615	0.5000
165	16	14	8	0.5000	0.5714
166	34	32	18	0.5294	0.5625
167	34	32	20	0.5882	0.6250
168	18	16	6	0.3333	0.3750
169	36	34	20	0.5556	0.5882
170	38	36	20	0.5263	0.5556
171	20	18	10	0.5000	0.5556
172	38	36	22	0.5789	0.6111
173	38	36	22	0.5789	0.6111
174	30	28	–	–	–
175	60	58	42	0.7000	0.7241
176	56	54	28	0.5000	0.5185
177	28	26	20	0.7143	0.7692
178	79	77	36	0.4557	0.4675
179	79	77	60	0.7595	0.7792
180	37	35	–	–	–
181	69	67	48	0.6957	0.7164
182	87	85	36	0.4138	0.4235
183	31	29	12	0.3871	0.4138
184	100	98	–	–	–
185	92	90	36	0.3913	0.4000
186	42	40	–	–	–
187	102	100	44	0.4314	0.4400
188	96	94	–	–	–
189	28	26	–	–	–
190	102	100	–	–	–
191	90	88	–	–	–
192	36	32	–	–	–
193	84	78	–	–	–
194	115	109	–	–	–
195	43	39	–	–	–
196	129	121	–	–	–
197	97	89	–	–	–
198	35	30	–	–	–
199	127	119	–	–	–
200	104	95	–	–	–

<i>n</i>	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
201	44	39	–	–	–
202	115	106	1	0.0087	0.0094
203	155	146	1	0.0065	0.0068
204	58	53	1	0.0172	0.0189
205	185	176	1	0.0054	0.0057
206	168	159	1	0.0060	0.0063
207	44	39	1	0.0227	0.0256
208	173	164	–	–	–
209	149	140	–	–	–
210	52	47	1	0.0192	0.0213
211	152	143	1	0.0066	0.0070
212	182	173	1	0.0055	0.0058
213	58	53	1	0.0172	0.0189
214	220	211	3	0.0136	0.0142
215	172	163	3	0.0174	0.0184
216	43	38	5	0.1163	0.1316
217	175	166	33	0.1886	0.1988
218	157	148	33	0.2102	0.2230
219	57	52	13	0.2281	0.2500
220	143	134	43	0.3007	0.3209
221	183	174	55	0.3005	0.3161
222	65	60	23	0.3538	0.3833
223	225	216	79	0.3511	0.3657
224	167	158	71	0.4251	0.4494
225	43	38	15	0.3488	0.3947
226	183	174	81	0.4426	0.4655
227	159	150	69	0.4340	0.4600
228	60	55	24	0.4000	0.4364
229	164	155	80	0.4878	0.5161
230	226	217	108	0.4779	0.4977
231	78	73	32	0.4103	0.4384
232	286	277	154	0.5385	0.5560
233	222	213	122	0.5495	0.5728
234	68	63	26	0.3824	0.4127
235	260	251	106	0.4077	0.4223
236	214	205	94	0.4393	0.4585
237	66	61	14	0.2121	0.2295
238	220	211	74	0.3364	0.3507

<i>n</i>	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
239	272	263	74	0.2721	0.2814
240	97	92	–	–	–
241	361	352	68	0.1884	0.1932
242	319	310	16	0.0502	0.0516
243	91	86	16	0.1758	0.1860
244	365	356	4	0.0110	0.0112
245	305	296	4	0.0131	0.0135
246	102	97	12	0.1176	0.1237
247	310	301	12	0.0387	0.0399
248	364	355	–	–	–
249	100	95	–	–	–
250	386	377	–	–	–
251	322	313	–	–	–
252	68	63	–	–	–
253	400	391	–	–	–
254	298	289	–	–	–
255	110	105	–	–	–
256	288	279	–	–	–
257	364	355	–	–	–
258	126	121	–	–	–
259	446	437	–	–	–
260	348	339	–	–	–
261	80	75	–	–	–
262	342	331	–	–	–
263	306	295	–	–	–
264	84	78	–	–	–
265	320	310	–	–	–
266	346	334	–	–	–
267	122	114	–	–	–
268	444	434	–	–	–
269	348	338	–	–	–
270	114	106	–	–	–
271	386	374	–	–	–
272	309	299	–	–	–
273	81	75	–	–	–
274	299	287	–	–	–
275	351	339	–	–	–
276	97	82	–	–	–

<i>n</i>	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
277	429	390	–	–	–
278	335	294	–	–	–
279	107	94	–	–	–
280	369	302	–	–	–
281	309	254	–	–	–
282	123	102	–	–	–
283	331	278	–	–	–
284	365	286	–	–	–
285	101	78	–	–	–
286	399	302	–	–	–
287	335	254	–	–	–
288	69	50	–	–	–
289	401	318	–	–	–
290	311	238	–	–	–
291	123	102	–	–	–
292	289	222	–	–	–
293	365	286	–	–	–
294	142	118	9	0.0634	0.0763
295	462	358	9	0.0195	0.0251
296	348	270	1	0.0029	0.0037
297	80	62	1	0.0125	0.0161
298	354	270	13	0.0367	0.0481
299	318	246	13	0.0409	0.0528
300	84	66	1	0.0119	0.0152
301	320	254	1	0.0031	0.0039
302	358	278	13	0.0363	0.0468
303	134	110	13	0.0970	0.1182
304	444	350	1	0.0023	0.0029
305	348	270	1	0.0029	0.0037
306	122	102	17	0.1393	0.1667
307	394	294	17	0.0431	0.0578
308	308	238	1	0.0032	0.0042
309	80	62	1	0.0125	0.0161
310	307	239	22	0.0717	0.0921
311	359	279	22	0.0613	0.0789
312	97	75	2	0.0206	0.0267
313	429	335	2	0.0047	0.0060
314	347	267	26	0.0749	0.0974

<i>n</i>	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
315	119	99	26	0.2185	0.2626
316	369	287	2	0.0054	0.0070
317	309	239	2	0.0065	0.0084
318	139	119	18	0.1295	0.1513
319	347	263	18	0.0519	0.0684
320	365	287	–	–	–
321	101	79	–	–	–
322	411	315	–	–	–
323	347	267	–	–	–
324	69	51	–	–	–
325	401	319	–	–	–
326	333	261	–	–	–
327	145	125	–	–	–
328	291	225	–	–	–
329	367	289	–	–	–
330	193	169	–	–	–
331	513	409	–	–	–
332	351	273	–	–	–
333	83	65	–	–	–
334	377	293	–	–	–
335	341	269	–	–	–
336	85	67	–	–	–
337	321	255	–	–	–
338	371	291	–	–	–
339	147	123	–	–	–
340	445	351	–	–	–
341	349	271	–	–	–
342	157	137	–	–	–
343	429	329	–	–	–
344	311	241	–	–	–
345	83	65	–	–	–
346	337	269	–	–	–
347	389	309	–	–	–
348	99	77	–	–	–
349	431	337	–	–	–
350	365	285	–	–	–
351	137	117	–	–	–
352	369	287	–	–	–

<i>n</i>	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
353	309	239	—	—	—
354	139	119	—	—	—
355	347	263	—	—	—
356	365	287	—	—	—
357	101	79	—	—	—
358	429	333	—	—	—
359	365	285	—	—	—
360	71	53	—	—	—
361	403	321	—	—	—
362	349	277	—	—	—
363	161	141	—	—	—
364	291	225	—	—	—
365	367	289	—	—	—
366	177	153	—	—	—
367	497	393	—	—	—
368	349	271	—	—	—
369	81	63	—	—	—
370	367	283	—	—	—
371	331	259	—	—	—
372	85	67	—	—	—
373	321	255	—	—	—
374	381	296	—	—	—
375	157	128	—	—	—
376	447	352	—	—	—
377	351	272	—	—	—
378	173	136	—	—	—
379	445	328	—	—	—
380	311	240	—	—	—
381	88	69	—	—	—
382	332	251	—	—	—
383	384	291	—	—	—
384	100	77	—	—	—
385	452	357	—	—	—
386	362	269	—	—	—
387	126	93	—	—	—
388	388	305	—	—	—
389	332	261	—	—	—
390	160	123	—	—	—

<i>n</i>	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
391	382	281	–	–	–
392	386	307	–	–	–
393	110	87	–	–	–
394	452	343	–	–	–
395	388	295	–	–	–
396	74	55	–	–	–
397	430	347	–	–	–
398	355	271	–	–	–
399	159	127	–	–	–
400	307	241	–	–	–
401	387	309	–	–	–
402	161	121	–	–	–
403	495	375	–	–	–
404	367	289	–	–	–
405	87	69	–	–	–
406	399	303	–	–	–
407	363	279	–	–	–
408	86	68	–	–	–
409	345	279	–	–	–
410	407	315	–	–	–
411	179	143	–	–	–
412	457	363	–	–	–
413	369	291	–	–	–
414	156	120	–	–	–
415	445	329	–	–	–
416	319	249	–	–	–
417	89	71	–	–	–
418	321	241	–	–	–
419	383	291	–	–	–
420	99	77	–	–	–
421	469	375	–	–	–
422	371	279	–	–	–
423	135	103	–	–	–
424	389	307	–	–	–
425	333	263	–	–	–
426	175	139	–	–	–
427	397	297	–	–	–
428	385	307	–	–	–

<i>n</i>	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
429	109	87	–	–	–
430	435	327	–	–	–
431	371	279	–	–	–
432	71	53	–	–	–
433	427	345	–	–	–
434	337	253	–	–	–
435	147	115	6	0.0408	0.0522
436	329	263	22	0.0669	0.0837
437	409	331	22	0.0538	0.0665
438	185	145	6	0.0324	0.0414
439	543	423	30	0.0552	0.0709
440	403	325	34	0.0844	0.1046
441	103	85	14	0.1359	0.1647
442	437	341	30	0.0686	0.0880
443	405	321	34	0.0840	0.1059
444	91	73	5	0.0549	0.0685
445	391	325	46	0.1176	0.1415
446	427	335	28	0.0656	0.0836
447	185	149	14	0.0757	0.0940
448	485	391	30	0.0619	0.0767
449	405	327	38	0.0938	0.1162
450	143	107	5	0.0350	0.0467
451	465	349	38	0.0817	0.1089
452	349	279	30	0.0860	0.1075
453	105	87	22	0.2095	0.2529
454	367	287	39	0.1063	0.1359
455	441	349	54	0.1224	0.1547
456	107	85	9	0.0841	0.1059
457	519	425	82	0.1580	0.1929
458	417	325	45	0.1079	0.1385
459	165	133	21	0.1273	0.1579
460	423	341	53	0.1253	0.1554
461	371	301	61	0.1644	0.2027
462	171	135	5	0.0292	0.0370
463	479	379	93	0.1942	0.2454
464	469	391	79	0.1684	0.2020
465	161	139	51	0.3168	0.3669
466	551	443	107	0.1942	0.2415

<i>n</i>	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
467	503	411	123	0.2445	0.2993
468	91	73	9	0.0989	0.1233
469	641	559	195	0.3042	0.3488
470	481	397	111	0.2308	0.2796
471	225	193	71	0.3156	0.3679
472	449	383	121	0.2695	0.3159
473	655	577	251	0.3832	0.4350
474	239	199	33	0.1381	0.1658
475	815	695	269	0.3301	0.3871
476	557	479	169	0.3034	0.3528
477	157	139	65	0.4140	0.4676
478	579	483	165	0.2850	0.3416
479	543	459	169	0.3112	0.3682
480	134	116	36	0.2687	0.3103
481	723	657	357	0.4938	0.5434
482	625	533	217	0.3472	0.4071
483	253	217	97	0.3834	0.4470
484	803	709	321	0.3998	0.4528
485	795	717	409	0.5145	0.5704
486	238	202	68	0.2857	0.3366
487	849	733	387	0.4558	0.5280
488	701	631	353	0.5036	0.5594
489	181	163	99	0.5470	0.6074
490	772	692	406	0.5259	0.5867
491	907	815	497	0.5480	0.6098
492	238	216	126	0.5294	0.5833
493	1233	1139	781	0.6334	0.6857
494	888	796	494	0.5563	0.6206
495	228	196	92	0.4035	0.4694
496	1068	986	662	0.6199	0.6714
497	888	818	558	0.6284	0.6822
498	270	234	102	0.3778	0.4359
499	902	802	512	0.5676	0.6384
500	1176	1098	718	0.6105	0.6539
501	288	266	168	0.5833	0.6316
502	1240	1132	710	0.5726	0.6272
503	1008	916	576	0.5714	0.6288
504	182	164	84	0.4615	0.5122

<i>n</i>	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
505	1342	1260	848	0.6319	0.6730
506	944	860	520	0.5508	0.6047
507	332	300	152	0.4578	0.5067
508	954	888	588	0.6164	0.6622
509	1258	1180	816	0.6486	0.6915
510	384	344	180	0.4688	0.5233
511	1540	1420	976	0.6338	0.6873
512	1204	1126	780	0.6478	0.6927
513	230	212	130	0.5652	0.6132
514	1129	1033	684	0.6058	0.6621
515	1005	921	604	0.6010	0.6558
516	315	297	162	0.5143	0.5455
517	1367	1301	818	0.5984	0.6287
518	1399	1307	848	0.6061	0.6488
519	445	409	242	0.5438	0.5917
520	2207	2113	1484	0.6724	0.7023
521	1661	1583	1034	0.6225	0.6532
522	425	389	128	0.3012	0.3290
523	1687	1571	994	0.5892	0.6327
524	1493	1423	858	0.5747	0.6030
525	369	351	274	0.7425	0.7806
526	1435	1355	868	0.6049	0.6406
527	1765	1673	1198	0.6788	0.7161
528	501	479	280	0.5589	0.5846
529	2543	2449	1894	0.7448	0.7734
530	1722	1630	1120	0.6504	0.6871
531	374	342	240	0.6417	0.7018
532	2230	2148	1516	0.6798	0.7058
533	1850	1780	1308	0.7070	0.7348
534	536	500	148	0.2761	0.2960
535	1776	1676	1136	0.6396	0.6778
536	2396	2318	1436	0.5993	0.6195
537	588	566	424	0.7211	0.7491
538	2426	2318	1436	0.5919	0.6195
539	2002	1910	1296	0.6474	0.6785
540	400	382	116	0.2900	0.3037
541	2664	2582	1848	0.6937	0.7157
542	1860	1776	1096	0.5892	0.6171

<i>n</i>	Total (SL 99)	Total (SL 23)	Errors	Err. ratio (SL 99)	Err. ratio (SL 23)
543	584	552	352	0.6027	0.6377
544	2333	2267	1028	0.4406	0.4535
545	2845	2767	1840	0.6467	0.6650
546	822	782	164	0.1995	0.2097
547	3690	3570	2192	0.5940	0.6140
548	3050	2972	1244	0.4079	0.4186
549	502	484	220	0.4382	0.4545
550	2810	2714	1076	0.3829	0.3965
551	2474	2390	924	0.3735	0.3866
552	654	636	–	–	–
553	2735	2669	1212	0.4431	0.4541
554	2925	2833	924	0.3159	0.3262
555	809	773	332	0.4104	0.4295
556	4395	4301	1468	0.3340	0.3413
557	3143	3065	1420	0.4518	0.4633
558	643	607	–	–	–
559	3063	2947	1220	0.3983	0.4140
560	2748	2678	876	0.3188	0.3271
561	504	486	180	0.3571	0.3704
562	2333	2253	–	–	–
563	2885	2793	684	0.2371	0.2449
564	805	783	–	–	–
565	3973	3879	1180	0.2970	0.3042
566	2719	2627	–	–	–
567	443	411	–	–	–
568	3519	3437	–	–	–
569	2723	2653	–	–	–
570	737	685	–	–	–
571	2717	2553	–	–	–