Some Studies On Information Set Decoding Algorithms and Universal Hash Functions

A thesis submitted to Indian Statistical Institute
in partial fulfillment of the thesis requirements for the degree of
Doctor of Philosophy in Computer Science

Author:
Sreyosi Bhattacharyya
bhattacharyya.sreyosi@gmail.com

Supervisor:
Prof. Palash Sarkar
palash@isical.ac.in

Applied Statistics Unit
Indian Statistical Institute
203, B. T. Road, Kolkata,
West Bengal, India - 700 108.

To my parents.

Acknowledgements

At the end of this six-year long enriching journey, I would like to thank everyone who has helped me during this period.

My supervisor, Prof. Palash Sarkar, has been very supportive and encouraging throughout the journey. His expert knowledge and academic advice have been of great help for me. I have learnt a lot from him. I am grateful to him for his unwavering support and valuable guidance.

I would like to express my gratitude towards my course teachers- Prof. Rana Barua, Prof. Palash Sarkar, Prof. Susmita Sur-Kolay, Prof. Mandar Mitra, Prof. Sasthi Charan Ghosh, Prof. Nabanita Das, Prof. Mridul Nandi, Dr. Arijit Bishnu, Dr. Debrup Chakraborty, and Dr. Sushmita Ruj. They have been our teachers during the M.Tech(CS) course at ISI, Kolkata. Dr. Sushmita Ruj has been very kind and encouraging. I had enjoyed interacting with her during the M.Tech course.

I wish to thank the ASU Office, the Dean's office and CSSC staff for their help in carrying out the official works.

I am extremely lucky to have very supportive seniors at our Turing Lab. Thanks to Sanjay-da (Dr. Sanjay Bhattacherjee), Butu-da (Dr. Shubhabrata Samajdar), Subhadip-da (Dr. Subhadip Singha), Sebati-di (Dr. Sebati Ghosh), Kaushik-da (Dr. Kaushik Nath), Aniruddha-da (Dr. Aniruddha Biswas) and Madhurima-di (Dr. Madhurima Mukhopadhyay). They have been extremely encouraging throughout these years. I would like to thank Aniruddha-da for listening to me patiently during some tough days. I would like to thank some of my peers- Debasmita, Susanta (Dr.Susanta Samanta), Jyotirmoy (Dr. Jyotirmoy Basak), Chandran, Debendra, Animesh, Rakesh and Sourav.

I am blessed to have some crazy friends- Subhra, Manish, Mohammad, Manobendra, Akanksha and Snehal-di. We have made many wonderful memories during these past years. Subhra has been very supportive and gentle towards me. I wish I can relive those sweet days with my friends again.

During the first few years of this journey I have enjoyed learning yoga from Mrs. Bijaya Saha at the Ladies' Hostel in ISI, Kolkata. I have been introduced to the martial art called Taekwondo at ISI, Kolkata by Senior Master Ruma Roy Chowdhury. I am grateful to her for being extremely kind to me when I used to struggle with the technicalities of the kicks, punches, blocks and footwork. Her extreme patience has helped me to learn and practice this great martial art and now I want to practice it throughout my life.

My parents have been my constant support system. Their unconditional love and constant emotional support have helped me to face all the challenges that have come in my way since my childhood. I am indebted to them. Moni's unconditional love for me have always made me feel special. She too has given me emotional support during these years.

Signature of Candidate with date

List Of Notations And Abbreviations of Part I

$\mathbb{F}_2$	:	A finite field with two elements
k	:	Dimension of a code $\mathcal{C}$
n	:	Length of the code $\mathcal{C}$
$\mathbf{H_0}$	:	Parity check matrix of $\mathcal{C}$
$\mathbf{I}_i$	:	Identity matrix of order i
$\mathbf{0}_i$	:	All zero vector of length i
$\mathbf{1}_i$	:	All one vector of length i
p	:	Number of ones allowed inside the information set
$\mathcal{I},\ \mathcal{I}_1,\ \mathcal{I}_2$	:	Information sets with $\#\mathcal{I}_1 = \#\mathcal{I}_2 = p/2$ and $\mathcal{I} = \mathcal{I} \cup \mathcal{I}_2$
$\mathbf{e}$	:	Error vector of length n
$\mathbf{s_0}$	:	Syndrome vector of length n
ℓ	:	Number of positions checked for collision in the meet-in-the-middle step
λ	:	An integer in $[-\ell, \ell]$
δ	:	A new parameter with value in $(0, 1]$
ω	:	Hamming weight of e
$(\mathbf{u}, \mathbf{v})$	:	$\mathbf{s} = (\mathbf{u}, \mathbf{v})$
ISD	:	Information Set Decoding

List of Notations And Abbreviations Of Part II

$\mathbb{F}_p$	:	prime order field
$\mathbb{Z}_p$	:	prime order field with elements $\{0, \cdots, p-1\}$
$\mathcal{K}$	:	A finite set called the Key space
$\mathcal{D}$	:	A finite set called the domain of the hash function
$\mathcal{R}$	:	A finite set called the range of the hash function
τ	:	$\tau \in \mathcal{K}$
X	:	Input string
M	:	Input string after padding
m	:	Number of bits required to represent an element of $\mathbb{Z}_p$
k	:	Number of bits in key
n	:	Number of bits per block of X
ℓ	:	Number of kn-bit blocks in X
$\mathbf{d}$	:	degree of decimation of Horner's method
$\mathsf{Poly}(x; M)$	:	Hash functions based on Univariate Polynomial which is evaluated using Horner's rule and x is the key and $M = (M_1, \cdots, M_\ell)$ is the padded input.
$\mathsf{BRW}(x; M)$	:	Hash functions based on BRW Polynomial and x is the key and $M = (M_1, \cdots, M_\ell)$ is the padded input.
$\mathsf{Poly1305}(x; M)$	:	Univariate polynomial defined over $\mathbb{F}_{2^{130}-5}$ as studied in Chapter 9

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The fundamental aims of Cryptography are to ensure *secure or safe communication* between two parties connected via an open/untrusted/public network and *safety* of information stored in a disk. The *safe communication* and *storage* must provide security against either any one or both of the following types of adversaries depending on real-world demands: passive or eavesdropping adversaries and active adversaries. The notion of a *secure communication* and *storage* entails the two following aspects.

- secrecy
- integrity

Secrecy secures the communication from passive adversaries while *integrity* secures the communication from active adversaries. Depending on practical scenarios a communication may need either any one of *secrecy* and *integrity* or both *secrecy* and *integrity*. These aspects of a *secure communication* and *secure storage* are fulfilled by each of the two broad areas of Cryptography, namely, Symmetric Key Cryptography and Public Key Cryptography. In symmetric-key settings symmetric-key encryption schemes ensure secrecy and Message Authentication Codes are used for ensuring integrity. In public-key settings public-key encryption schemes provide secrecy while signature schemes are used to ensure integrity.

Symmetric Key Cryptography

Any symmetric key encryption system is defined over three finite sets: key space $\mathcal{K}$, message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ and it consists of three algorithms: the key generation algorithm, encryption algorithm and decryption algorithm. The key generation algorithm is a probabilistic polynomial-time algorithm which takes some security parameter as input and outputs a secret key $sk \in \mathcal{K}$. The encryption algorithm is, again, probabilistic polynomial-time. It takes the plaintext $m \in \mathcal{M}$ and the secret key sk as inputs and outputs a ciphertext $c \in \mathcal{C}$. The decryption algorithm is deterministic which takes c and sk as inputs and returns a plaintext. The algorithms must satisfy the correctness property such that m and $Decryption_{sk}(Encryption_{sk}(m))$ are same. There are two types of ciphers, the primitives

used for encrypting a message using a shared secret key, namely stream cipher and block cipher. In the symmetric-key setup the shared secret key sk must be known to the sender and receiver before the beginning of the communication.

Encryption schemes provide *confidentiality* while message *integrity* is achieved through the use of Message Authentication Codes (MAC). Sometimes only *integrity* is expected from a *safe communication* over an open channel. For example we may consider the following scenarios. Suppose Alice is expecting a message from Bob but an adversary Eve sends the message instead of Bob. With the help of MAC Alice can check whether the received message is actually sent by Bob. Now, let us suppose that Bob sends a message to Alice but Eve tampers the message and sends this new message to Alice. Again, Alice can find out whether the received message is the original message sent by Bob. A MAC needs three polynomial-time algorithms: key generation algorithm, tag generation algorithm and verification algorithm. The tag generation algorithm takes message m and secret key sk as inputs and outputs a tag t. Alice runs the tag generation algorithm using m and sk and sends the message-tag pair (m, t) to the receiver. The receiver Bob verifies the tag by running the verification algorithm. The verification algorithm takes (m, t, sk) and computes the tag t' with the shared secret key sk and received message. The verification succeeds if t and t' are same. Thus message integrity can be verified using MACs. Sometimes both secrecy and integrity are needed. In such cases authenticated encryption schemes are used. If some additional data which needs only authentication requires to be sent along with message that requires both authentication and encryption then Authenticated Encryption with Associated Data (AEAD) schemes are used.

Universal Hash Function. Universal hash functions are one of the primitives which are used to construct MACs. Universal hash functions can be built using matrix multiplication, inner product, polynomial evaluation etc. Constructions using univariate polynomials have been used widely in real life. This may be due to the simplicity in their designs and implementations. Polynomials defined over binary extension fields and prime order fields are two candidates for instantiating universal hash functions. The indeterminate takes value from the key space. An important example is Poly1305, proposed by Bernstein[16]. The MAC described in [16] used Poly1305 along with AES. In case of polynomials defined over prime order fields, the structure of the prime number can be highly exploited to have efficient implementations of the hash functions based on evaluation of such polynomials. Padding schemes play an important role in the security of such hash functions. The padding scheme should be chosen carefully otherwise there may be successful forgery attack. The security of these universal hash functions include two notions: *collision probability* and *differential probability*. If collision probability is upper bounded by ϵ then such a hash function is called ϵ-almost universal function. If differential probability is upper bounded by ϵ then such a function is called ϵ-almost XOR universal hash function. This ϵ must be negligible.

For building univariate polynomial evaluation based hash functions, the input message is divided into, say, ℓ number of blocks of equal length (except the last block) and such blocks form the coefficients of the polynomial. These ℓ number of blocks can be used to build a degree-$(\ell - 1)$ polynomial. The polynomial needs to be evaluated at a point in the relevant

finite-field. Field multiplications are required for this purpose. The multiplication routines are very costly in terms of CPU cycles. So reducing the number of field multiplications or speeding up the multiplication routines are important. A simple but efficient way to evaluate such univariate polynomials is Horner's method. Horner's method requires $\ell - 1$ multiplications and $\ell - 1$ additions for evaluating such polynomials of degree $\ell - 1$ built with ℓ blocks. The multiplication routine of Horner's rule based evaluations are made to perform better using SIMD evaluations. Intel's AVX2 intrinsics have been used for this purpose in [60]. An alternative way for speeding up the multiplication routines is to reduce the number of invocations of the reduction routine. Apart from Horner's rule another method of polynomial evaluation is the evaluation technique of BRW polynomials. A BRW polynomial with ℓ blocks require $\lfloor \ell/2 \rfloor$ multiplications and $\lfloor \log_2 \ell \rfloor$ squarings. These polynomials are attractive candidates in terms of lesser number of multiplications. Though definition of BRW polynomials is inherently recursive, the work in [59] gives non-recursive algorithm for carrying out the evaluation. So the BRW polynomials form potential candidates to be tried for instantiating polynomial evaluation based universal hash functions.

Universal hash functions are also used to construct tweakable enciphering schemes. MACs based on polynomial evaluation-based hash functions are also used in AEAD schemes. AES-GCM and ChaCha20-Poly1305 are two AEAD schemes which have been used in real life applications.

Classical Public Key Cryptography and The Need For Post-Quantum Public Key Cryptosystems

In the year 1976 Whitfield Diffie and Martin Hellman came up with a seminal approach [43] of exchanging keys safely over an open network. The keys that are exchanged via such a network are called public-keys. The public-keys are computed based on a trapdoor one-way function such that it is simple to compute the public-key using the private-key but it is computationally intractable to invert the function without the knowledge of the private key. More specifically, Diffie and Hellman [43] uses the one-way function $b = f(a) = g^a$. It is easy to compute if the trapdoor a is known but it is computationally difficult to retrieve a from b. Each of the parties keeps the secret/private key only with itself. The main idea is to instantiate the one-way trapdoor function using a computationally intractable problem. The Diffie-Hellman key exchange protocol uses discrete logarithm problem to serve this purpose and thus security of the Diffie-Hellman Key Exchange Protocol depends on the intractability of discrete logarithm problem. The sender Alice generates a description of a cyclic group $\mathbb{G}$, generator g of $\mathbb{G}$, an integer q and chooses $x \in \mathbb{Z}_q$ and computes g^x. Then it sends g, g^x to the receiver Bob. Bob in turn chooses $y \in \mathbb{Z}_q$, computes g^y and sends g^y to Alice. Bob computes $(g^x)^y$ and Alice computes $(g^y)^x$. Thus the shared secret key known only to Bob and Alice is g^{xy}.

The first Public-key Encryption scheme was proposed by Ron Rivest, Adi Shamir and Len Adleman [93] in the year 1978. This is called the RSA Algorithm after the name of its inventors. In a PKE the sender encrypts the message using the public key of the receiver and the receiver upon the receipt of the 'encrypted' message, called the ciphertext, uses its

secret key to decrypt the received ciphertext. The security of the RSA cryptosystem relies on the 'believed' one-wayness of integer factorization problem. The receiver Bob chooses two prime integers p and q and integers a and b such that $a \cdot b \equiv 1 \bmod (p-1) \cdot (q-1)$ and shares (N, a) where $N = p \cdot q$ with Alice. Thus (N, a) forms the public key and $p,\ q$ and b are the secrets. The sender encrypts message as $c = m^a \bmod N$. The receiver decrypts as $c^b \bmod N$. Though there are many attacks on this plain version of RSA it is a seminal work and thus important to the Cryptology community. Apart from the RSA algorithm some other major PKE are the ElGamal PKE and Elliptic-curve based PKE.

On the other hand, digital signature scheme is used to get message integrity in a public-key setup. Any digital signature scheme consists of three algorithms: key generation ($keygen$), signing algorithm ($sign$) and verification algorithm (ver). The key generation algorithm generates one secret key sk and one public key pk. The sender runs the signing algorithm ($sign$) with the message and secret key as its inputs to produce the signature s as $sign(m, sk)$. It sends to the receiver (m, s) and the receiver verifies the signature using the public key of the sender such that the verification is successful if $ver(pk, m, s)$ outputs *accept*. The RSA and ElGamal signature schemes are two well researched digital signature schemes. The former is based on the RSA assumption and the latter is based on the difficulty of computing discrete logarithm.

All of these describe the importance of the notion of Public-Key Cryptography in the modern days.

The major forms of security of any PKE/KEM scheme include the security against:

- Chosen Plaintext Attack (CPA-security),
- Chosen Ciphertext Attack (CCA-1/CCA-2-security).

The major form of security of any digital signature scheme include the security against existential forgery under chosen message attack. These security notions of encryption/KEM schemes and signature schemes require the one-way feature of a trapdoor function as a necessary condition. As mentioned earlier, the 'one-way' feature makes such functions easy to compute but (computationally) difficult to invert without the knowledge of the trapdoor which is a secret. The computationally intractable problems like prime factorization of integers, discrete logarithm have been used to instantiate such trapdoor functions and thus these problems help to build widely deployed *secure* public-key encryption schemes. For example, security of ElGamal cryptosystem and elliptic-curve-based cryptosystems depend on the believed intractability of discrete-logarithm problem for multiplicative cyclic group and elliptic curves respectively.

In the year 1994, Peter Shor proposed a quantum algorithm [98] which can solve the prime factorization problem and discrete logarithm problem in polynomial time using phase estimation algorithm. Shor reduced the integer factorization problem to order-finding problem which can be solved in polynomial time by the quantum algorithm. He argues that if an integer n is odd and additionally it is not an odd power then we choose a random integer x such that $2 \leq x < n$. If $gcd(x, n) = 1$ then no non-trivial factor x of n is found. Then we can use a quantum algorithm which computes r such that $x^r \equiv 1 \bmod n$. The quantum algorithm does the phase-estimation followed by an invocation of continued fraction algorithm

to compute r. The only constraint over the value of r is that r must be even because once r has been computed we can find $gcd(n, x^{r/2}+1)$. Thus the other factor is $n/gcd(n, x^{r/2}+1)$. As a consequence the PKE/KEM schemes widely in practice are broken. Examples of such systems are RSA-3072, DH-3072, 256-bit ECDH etc. This explains the need for studying the cryptosystems that are based on computationally intractable problems which cannot be solved efficiently by quantum computers too. Such problems can be used to instantiate trapdoor one-way functions to build quantum-safe KEM, PKE and digital signature schemes. One of such problems is the decoding of random linear error-correcting codes. Other problems include various lattice problems- Shortest Vector Problem, Shortest Independent Vector Problem, Learning With Errors Problem etc. and the problem of solving non-linear equations over finite fields. Post-quantum Cryptography involves study of classical cryptographic algorithms which are secure even against attacks by quantum adversaries.

The Post-quantum Cryptography competition of NIST which started in 2016 announced the first winner of the standardization process on July 5, 2022. The winning algorithm in the encryption/KEM category was CRYSTALS_kyber [8] which relies on the hardness of module Learning with Error problem.

Classic McEliece [19], BIKE[4] and HQC [83] are three candidates from the code-based cryptography category in the fourth round [88] of the standardization process. There can be different types of code-based cryptosystems depending on the type of distance metric being considered. The distance metric studied widely are hamming metric, Lee metric and rank metric. All the three candidates of the fourth round of the NIST PQC competition consider the hamming metric.

Code-based Cryptography. In code-based cryptography [89] the underlying one-way function uses an error-correcting code and involves either of the following.

- encoding a message into a codeword of the code and subsequent addition of random error vector to the codeword
- treating the message as error vector and computation of a syndrome vector using the parity-check matrix of the code.

Of the two techniques mentioned above, the first one is used in McEliece public key encryption scheme [80] and the second one is used in Niederreiter public key encryption scheme [86]. The trapdoor in McEliece PKE is the generator matrix of the code being used and that in Niederreiter PKE is the parity-check matrix corresponding to the code being used. Apart from the secret code, the private-key in both of the encryption schemes also includes a permutation matrix and a scrambler matrix.

The main advantages of code-based encryption schemes are fast encryption and decryption algorithms. On the downside, some of the encryption schemes require large keys.

The first public-key encryption scheme based on error-correcting codes was proposed by McEliece [80] in the year 1978. This PKE used binary irreducible Goppa-code as the underlying error-correcting code. This scheme has no structural attacks till date. If the ciphertext for a random input obtained from the original McEliece scheme and the public key are available then it is difficult to get the input message (this has been termed as oneway CPA

or OW-CPA) but it is not secured against IND-CCA-2 attacks. But it can be transformed to CCA-2 secured version [69]. Later in 1986 another PKE was proposed by Niederreiter [86] with General Reed-Solomon code but it was broken by Sidelnikov & Chestakov [106]. A number of encryption schemes based on Concatenated code, Reed-Muller code, convolutional LDPC codes were proposed but all of them were subsequently broken by structural attacks. Apart from encryption schemes several works based on error-correcting codes have been proposed. For example, an identification scheme by Stern [102], a PRG by Fischer et al. in [56], hash functions by Augot et al. in [7], the CFS signature scheme in [37].

The security of code-based cryptosystems relies on the hardness of decoding of random linear codes. Under the assumption that no algebraic structure of the code is revealed to the adversary, the adversary can retrieve the message/plaintext from the corrupt codeword only by solving the *decoding problem*. Or if the syndrome vector is known, the adversary can retrieve the error vector only by solving the *Computational Syndrome Decoding* problem. If an adversary has any knowledge about the algebraic structure of the secret code then it can carry out structural attacks on the secret code. The most powerful technique of cryptanalysis when no information about the structure of the underlying code can be used to attack an encryption scheme, has been the Information Set Decoding (ISD) technique. Consequently, ISD technique form the best known attack against the McEliece encryption scheme.

Information Set Decoding. An ISD algorithm does not exploit the structure of the underlying code. It tries to construct an error vector of low-weight with the knowledge of the public parity-check matrix and syndrome vector. The first ISD algorithm dates back to 1962 which was proposed by Prange [91]. Ever since there have been quite a few attempts to improve the ISD technique in Leon [77], Lee-Brickell [76], Stern [101], Dumer [46], Finiasz-Sendrier [55], Bernstein et al. [38], MMT [78], BJMM [10], May-Ozerov [79], Both-May [25], Carrier et al. [31] etc. One of the cornerstones of ISD algorithms is the Stern's attack [101] which was further improved by Dumer [46]. Both of these attacks used meet-in-the-middle technique to obtain a solution. The next novel approach was to use the representation technique introduced by Howgrave-Graham and Joux [65] in MMT [78] and in BJMM [10]. Further advanced algorithms were proposed by May and Ozerov in [79] and by Both and May in [25]. In 2022 Carrier et al. proposed a new approach [31] of statistical decoding which reduces decoding problem to Learning Parity with Noise problem (LPN).

ISD algorithms are heuristic in nature. Such algorithms solve an instance of the *Computational Syndrome-Decoding Problem* which is a hard problem. Thus these algorithms take exponential time in terms of the code length to provide a solution to an instance of *Computational Syndrome-Decoding Problem*. For a given parity-check matrix $\mathbf{H_0}$ of dimension $(n-k) \times n$, a vector $\mathbf{s_0}$ of length $(n-k)$ and a positive integer ω any ISD aims to obtain an error vector $\mathbf{e}$ of length n and hamming weight ω such that

$$\mathbf{H_0} \cdot \mathbf{e}^\intercal = \mathbf{s_0}^\intercal.$$

Any ISD algorithm has the following three components which contribute to the total time complexity.

- expected number of iterations

- expected cost of linear algebra to obtain the systematic form of the parity-check matrix
- expected number of bit operations related to enumeration and matrix-vector multiplications per iteration

These algorithms guess k number of positions of the error vector $\mathbf{e}$ to be free of errors. This set of error-free positions is called the *Information Set*. So the errors are present in the rest of the $(n-k)$ positions outside the information set and number of combinations to be tested for the equation $\mathbf{H_0} \cdot \mathbf{e}^\intercal = \mathbf{s_0}^\intercal$ is $\binom{n-k}{\omega}$. The columns of $\mathbf{H_0}$ corresponding to the one positions are added up. This sum vector is then checked against the syndrome vector $\mathbf{s_0}^\intercal$. This is the Prange's attack. All the subsequent works allow a small number of error positions inside the *information set*.

Without guessing a set of error-free positions, we have to naively search $\binom{n}{\omega}$ combinations. So due to the guessing of a set of k error-free positions, the size of the search space is reduced to $\binom{n-k}{\omega}$. Let the guess be correct with probability π. If each iteration is independent then the expected number of iterations required to hit a success is $1/\pi$. Please note that π depends on the particular algorithm used to obtain the error vector.

The number of field operations done per iteration is determined by the number of combinations of error vectors enumerated and the cost of adding up the one positions of each of those enumerations.

The timeline of ISD algorithms has started with the Prange's attack which needs $O(1)$ memory. The latter advanced algorithms give improvement in terms of asymptotic complexity of time but the memory complexity worsens as exponential number of enumerations are stored. This scenario also suggests to study time/memory trade-off of ISD algorithms.

Thesis Overview

This thesis has two parts. One part studies Information Set Decoding Algorithms. The other one studies universal hash functions defined over prime order fields.

1.1 Overview of Part I

This part of the thesis focuses on Information Set Decoding (ISD) which is the best known attack technique against code-based encryption schemes when no knowledge about the structure of the underlying code is available. Chapter 2 gives relevant preliminaries and notations. In Chapter 3 we give a brief survey of literature focusing on ISD algorithms and code-based candidates of NIST PQC standardization competition.

Chapter 4 describes our work which gives a generalised version of the Stern's attack. Stern's algorithm, a variant of Stern's algorithm due to Dumer, as well as a recent generalisation of Stern's algorithm due to Bernstein and Chou are obtained as special cases of our generalisation. The generalisation is achieved due to the introduction of two new parameters λ and δ. The parameter λ takes integer values from the $[-\ell, \ell]$ and $\delta \in (0, 1]$. So our generalised algorithm needs two more parameters besides the usual parameters p and ℓ

required by Stern's algorithm. The information set is of size k for Stern's algorithm while the information set is of size $k + \ell$ in case of Dumer's algorithm. Both of the algorithms allow p error positions inside the information set and rest of the $\omega - p$ errors are outside the information set. To construct the error vector both of the algorithms divide the information set into two halves. Stern's algorithm and Dumer's algorithm enumerate $\binom{k/2}{p/2}$ combinations and $\binom{(k+\ell)/2}{p/2}$ combinations respectively from each of the halves. The error vector constructed by Stern's attack has a run of zeroes of length ℓ while the error vector obtained by Dumer's attack does not have any run of zeroes. Our algorithm needs information set of size $k + \lambda$ and the length of the run of zeroes is $\ell - \lambda$. This generalised version enumerates $\binom{(k+\lambda)/2}{p/2}$ combinations from one half and $\binom{(k+\lambda)/2}{p/2}^{\delta}$ combinations from the other half. We have studied the performance of the generalisation using the five variants of Classic McEliece considered in the NIST PQC standardization project.

Chapter 5 introduces the notion of a set of effective time/memory trade-off (TMTO) points for any ISD algorithm for the considered ranges of values of parameters of the algorithm. Such a set succinctly and uniquely captures the entire landscape of TMTO points with only a minor loss in precision. We further describe a method to compute a set of effective TMTO points. As an application, we compute sets of effective TMTO points corresponding to the new algorithm in Chapter 4 as well as for Stern's and Bernstein-Chou algorithms for the five variants of the Classic McEliece cryptosystem. The results show that while Dumer's and Bernstein and Chou's algorithms do not provide any interesting TMTO points beyond what is achieved by Stern's algorithm, the new generalisation that we propose provides about twice the number of effective TMTO points that is obtained from Stern's algorithm. Consequences of the obtained TMTO points to the classification of the variants of Classic McEliece in appropriate NIST categories are discussed.

1.2 Overview of Part II

This part of the thesis focuses on universal hash functions defined over prime order fields. Chapter 7 gives the required preliminaries and notations. In Chapter 8 we give a brief survey of literature focusing on universal hash functions.

Chapter 9 proposes an improved SIMD implementation of Poly1305 which is a polynomial-evaluation based hash function designed by Bernstein [16]. It is an univariate polynomial-evaluation based hash function which is defined over the prime order field $\mathbb{F}_{2^{130}-5}$. It is deployed widely in real-life applications including the Transport Layer Security. The univariate polynomial used for the construction is a degree ℓ polynomial where ℓ depends on the length of the input message. Horner's method is an efficient option for the evaluation of the polynomial. The structure of Horner's method is suitable for a vectorized or SIMD evaluation. Vectorized implementation of Poly1305, using Intel's AVX2 intrinsics [66], has been proposed by Goll and Gueron in [60] which uses 4-way SIMD evaluation but this 4-way SIMD is not done throughout the length of the message when the number of 16-byte blocks is not a multiple of four. It is applied only up to the point such that the number of 16-byte blocks in the message is a multiple of four. So for the rest of the evaluation 256-bit packed

representation requires to be unpacked. As Poly1305 has been in use in TLS, SSH etc it is interesting to study its performance if the 4-way SIMD is used for the entire length of the message. We propose a simple *balancing technique* which leads to the improvements of Goll-Gueron vectorization strategy. This balancing of the 4-way vectorization is achieved by prepending certain number of zeroes to the message. This does not increase the length of the message because the zeroes are prepended logically. If the message does not have number of 16-byte blocks in multiples of four then there are three possible cases- prepend three 16-byte blocks of zeroes, prepend two 16-byte blocks of zeroes and prepend one 16-byte block of zeroes. We also propose new initial multiplication algorithms for each of the cases when one, two and three blocks of zeroes are prepended to the message. Comparison between the performances of the two versions have been done in Intel's Haswell, Skylake and Kaby Lake micro-architectures. Several compilers have been considered too. The range of message lengths considered for this comparison is 49 bytes to 4000 bytes. The comparisons show marked improvements in speed for short messages. Both of the Goll-Gueron implementation and the improved implementation are done using Intel's AVX2 Intrinsics.

In Chapter 10 constructions of new universal hash functions have been done. Polynomial evaluation-based hash functions defined over prime order fields are important for building universal hash functions. Such polynomials are evaluated following Horner's rule or evaluation of BRW polynomials. A degree $\ell - 1$ polynomial with ℓ blocks can be evaluated at a point using $\ell - 1$ field multiplications while a BRW polynomial needs $\lfloor \ell/2 \rfloor$ multiplications and $\lfloor \ell/4 \rfloor + 1$ reductions (due to [59]) with additional $\lfloor \log_2 \ell \rfloor$ number of squarings. Before this work BRW polynomials have been studied only in binary extension fields. Reducing the number of field multiplications is important because it is a costly operation. So it is interesting to study the performance of BRW polynomials over prime order fields. This chapter makes a comprehensive study of the said two important strategies for polynomial hashing over a prime order field $\mathbb{F}_p$ and several combinations of them.

For each of the hash functions, we propose concrete instantiations using the primes $2^{130}-5$ and $2^{127} - 1$. Upper bounds on hash functions' differential probabilities have been derived. We propose extensive 64-bit implementations of all the proposed hash functions in assembly targeted at modern Intel processors. Each element of $\mathbb{F}_{2^{130}-5}$ can be represented using three 64-bit registers and each element of $\mathbb{F}_{2^{127}-1}$ can be represented using two 64-bit registers. Multiplication routines of $\mathbb{F}_{2^{127}-1}$ are faster than that of $\mathbb{F}_{2^{130}-5}$. On the other hand, number of blocks for the same input message is greater for the finite field $\mathbb{F}_{2^{127}-1}$ than that for $\mathbb{F}_{2^{130}-5}$. The timing results suggest that hash functions using the prime $2^{127}-1$ are significantly faster than using the prime $2^{130}-5$. Performances of our implementations show that our final hash function proposal for the prime $2^{127} - 1$ is much faster than the well known Poly1305 hash function defined over the prime $2^{130} - 5$, achieving speed improvements of up to 40%. Hash functions based on evaluation of BRW polynomials are expected to be much faster than hash function based on Horner-based evaluation of polynomials because the former needs half the number of field multiplications than the latter. This is true for both $\mathbb{F}_{2^{127}-1}$ and $\mathbb{F}_{2^{130}-5}$ when the usual Horner's method is used but for grouped Horner the speed-up of BRW evaluation is modest. We explain the reason behind such substantial speed-up of grouped Horner based evaluation with respect to BRW evaluation.

Publications. This thesis is based on the following accepted papers.

1. Sreyosi Bhattacharyya and Palash Sarkar. Concrete Time/Memory Trade-Offs in Generalised Stern's ISD Algorithm. **Proceedings of Indocrypt 2023**, Lecture Notes in Computer Science, Springer, 307–328.

2. Sreyosi Bhattacharyya and Palash Sarkar. Improved SIMD implementation of Poly1305. **IET Information Security**, volume 14, number 5, 521-530, 2020, `https://doi.org/10.1049/iet-ifs.2019.0605`

3. Sreyosi Bhattacharyya, Kaushik Nath and Palash Sarkar. Polynomial Hashing over Prime Order Fields. **Advances in Mathematics of Communications**, `https://www.aimsciences.org/article/id/65a63bba92d3ad47ddcdd5f0`, 2024

Part I

Study on ISD Algorithms

Chapter 2

Preliminaries

The aim of this chapter is to give the relevant preliminaries required to understand Information Set Decoding algorithms. To serve this purpose we describe the relevant concepts of linear error-correcting codes, two basic code-based encryption schemes and the broad steps of an ISD algorithm.

2.1 Notations

Throughout this part vectors are considered as row vectors denoted by bold lower case letters. Matrices are denoted by bold upper case letters. Identity matrices are denoted by $\mathbf{I}_j$ where j is the order of the matrix. All-zero and all-one vectors of length j will be denoted as $\mathbf{0}_j$ and $\mathbf{1}_j$ respectively where j is a positive integer. The cardinality of any finite set $\mathcal{S}$ will be denoted as $\#\mathcal{S}$. The distance metric considered is the hamming metric, the hamming weight of any vector $\mathbf{v}$ is denoted as $|\mathbf{v}|$. For any positive integer i, $[i]$ denotes the set $\{1, \ldots, i\}$. If two matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ have equal number of rows then by $[\mathbf{M}_1|\mathbf{M}_2]$ we will denote the matrix obtained by juxtaposing $\mathbf{M}_1$ and $\mathbf{M}_2$. For any vector $\mathbf{v}$ of length j, $\mathbf{v}_{[i]}$ such that $i < j$ denotes the projection of the positions of $\mathbf{v}$ corresponding to $[i]$.

2.2 Some Relevant Definitions

Let n and k be two positive integers such that $n > k$ and $\mathcal{C}$ be a subspace of dimension k of a vector space $\mathcal{V} = \{\mathbf{x} : \mathbf{x} \in \mathbb{F}_2^n\}$, where $\mathbb{F}_2$ is a finite field of two elements. Let the basis vectors of $\mathcal{C}$ be given by a matrix $\mathbf{G}_{\mathbf{0}k\times n}$. Let $\mathbf{H}_{\mathbf{0}(n-k)\times n}$ be another matrix such that $\mathbf{G_0} \cdot \mathbf{H_0}^\intercal = \mathbf{0}^\intercal$. So $\mathcal{C}$ can be expressed in two ways: $\mathcal{C} = \{\mathbf{x}\cdot\mathbf{G_0} : \mathbf{x} \in \mathbb{F}_2^k\}$ and $\mathcal{C} = \{\mathbf{y} : \mathbf{y} \in \mathbb{F}_2^n,\ \mathbf{H_0}\cdot\mathbf{y}^\intercal = \mathbf{0}^\intercal\}$. Let d be the minimum hamming distance between any two vectors of $\mathcal{C}$. The matrices $\mathbf{G_0}$ and $\mathbf{H_0}$ are called the generator and parity-check matrices of $\mathcal{C}$ respectively.

Linear Code $\mathcal{C}(n,\ k,\ d)$ is a linear code over $\mathbb{F}_2$ of length n, dimension k and minimum distance d with $\mathbf{G}_{\mathbf{0}k\times n}$ as the generator matrix and $\mathbf{H}_{\mathbf{0}(n-k)\times n}$ as the parity-check matrix. A codeword $\mathbf{y} \in \mathcal{C}$ is obtained from $\mathbf{x} \in \mathbb{F}_2^k$ as $\mathbf{y} = \mathbf{x} \cdot \mathbf{G_0}$.

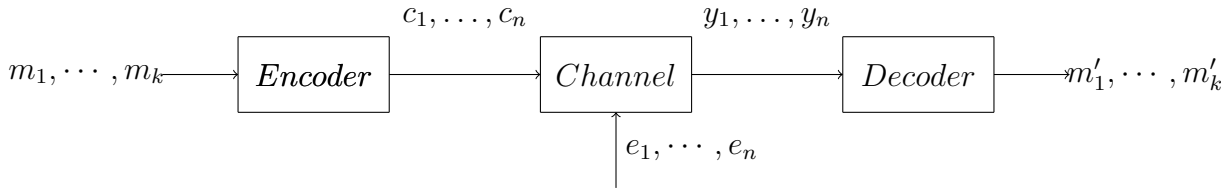


Figure 2.1: Communication over a noisy channel

The purpose of using codes is to detect and correct errors that may have happened during communicating a message over a noisy channel. Common examples of such noisy channels may be binary or q-ary symmetric channel, binary or q-ary erasure channel. The source generates a message $\mathbf{m}= (m_1, \ldots, m_k)$ and encodes it to $\mathbf{c}= (c_1, \ldots, c_n) \in \mathcal{C}$. Then, while in transit over the noisy channel $\mathbf{c}$ may have been changed to $\mathbf{y}$ such that $\mathbf{y} = \mathbf{c} \oplus \mathbf{e}$, where $\mathbf{e}^\intercal \in \mathbb{F}_2^n$ is a random error. The decoder at the receiving end checks whether the codeword has been received without any error. The decoder outputs $(m'_1, \ldots, m'_k)$ as the estimates of the actual message bits.

Syndrome Vector of y Any vector $\mathbf{y}$ is a valid codeword of $\mathcal{C}$ i.e., $\mathbf{y} \in \mathcal{C}$ iff the following equation is satisfied.

$$\mathbf{H_0} \cdot \mathbf{y}^\intercal = \mathbf{0}^\intercal \tag{2.1}$$

Otherwise,

$$\mathbf{H_0} \cdot \mathbf{y}^\intercal = \mathbf{H_0} \cdot (\mathbf{x} + \mathbf{e})^\intercal = \mathbf{H_0} \cdot \mathbf{e}^\intercal$$

$$\mathbf{H_0} \cdot \mathbf{e}^\intercal = \mathbf{s_0}^\intercal \ , \text{ where } \mathbf{s_0}^\intercal \in \mathbb{F}_2^{n-k}. \tag{2.2}$$

Here $\mathbf{s_0}$ is called the syndrome vector. In (2.1) $\mathbf{s_0}^\intercal = \mathbf{0}_{n-k}^\intercal$. Equation 2.2 may be rewritten as

$$\sum_{e_i \neq 0} \mathbf{H_0}_i = \mathbf{s_0}^\intercal \ , \text{ where } \mathbf{s_0}^\intercal \in \mathbb{F}_2^{n-k}. \tag{2.3}$$

Systematic Form of Parity Check Matrix. The parity check matrix is said to have a systematic form when it is expressed as

$$\mathbf{H_0} = [\mathbf{H}'|\mathbf{I}_{n-k}].$$

Given $\mathbf{H_0}$ the systematic form can be obtained through Gaussian elimination. The Gaussian elimination may be interpreted as multiplication of $\mathbf{H_0}$ by a $(n-k) \times (n-k)$ non-singular matrix $\mathbf{U}$. Additionally the columns chosen for the columns of the identity matrix may be chosen by an $n \times n$ permutation matrix $\mathbf{P}_n$. The entire process can be represented as

$$[\mathbf{H}'|\mathbf{I}_{n-k}] = \mathbf{U} \cdot \mathbf{H_0} \cdot \mathbf{P}.$$

Code Rate The code rate (R) of $\mathcal{C}$ is given by k/n where $R \in [0, 1]$.

Decoding The aim of decoding is to output a k-bit message which is closest in terms of hamming distance to the actual message. The output message can be uniquely determined if the number of error(s) is within a limit.

Error Correction and Error Detection If d is the minimum distance of a code $\mathcal{C}$ and it is odd i.e., of form $d = 2 \cdot t + 1$ then $\mathcal{C}$ can correct t errors. If d is even (of form $d = 2 \cdot t$) then it can correct $t - 1$ errors and only detect t errors. If the number of errors is more than the said numbers then the codewords cannot be determined uniquely. This explains why $(m'_1, \ldots, m'_k)$ of figure2.1 are only estimates of the input bits.

The computational problem to be solved to obtain the error vector for a received word is defined as follows.

Computational Syndrome Decoding Problem Let n, k and ω be three positive integers such that $n > k$. Let $\mathbf{H_0}_{(n-k)\times n}$ be a matrix such that $\forall i \in \{1, \ldots, n-k\}$ and $j \in \{1, \ldots, n\}$ $H_{ij} \in \mathbb{F}_2$ and $\mathbf{s_0}^\intercal \in \mathbb{F}_2^{n-k}$. The goal is to find $\mathbf{e}^\intercal \in \mathbb{F}_2^n$ such that $\mathbf{H_0} \cdot \mathbf{e}^\intercal = \mathbf{s_0}^\intercal$ and $|\mathbf{e}| = \omega$. The triplet $(\mathbf{H_0}, \mathbf{s_0}, \omega)$ forms an instance of *Computational Syndrome Decoding Problem*.

For arbitrary linear code Syndrome Decoding is a hard problem and the decision version of Syndrome Decoding Problem is NP-Complete [11].

Half and Full Distance Decoding If the weight of the error vector is allowed to be at most t then its is called half-distance decoding. If the weight error vector is allowed to go up to d then it is called full-distance decoding. For cryptographic purpose half-distance decoding is useful.

Gilbert-Varshamov bound(d') For given n and k, Gilbert-Varshamov bound is the largest integer d' such that $\sum_{i=0}^{d'-1} \binom{n}{i} \leq 2^{n-k}$.

2.3 Two Basic Code-based Cryptosystems

The McEliece and Niederreiter Cryptosystems are two classic examples of Code-based Cryptography. The original McEliece Cryptosystem [80] uses binary irreducible Goppa code, an error-correcting code, in the underlying trapdoor function. Another important work is the Niederreiter Cryptosystem which uses Generalized Reed-Solomon codes. Though it is broken the basic concept is useful. Two of the candidates in the fourth round of NIST PQC standardization competition follow the McEliece Cryptosystem.

The underlying one-way function of the McEliece Cryptosystem adds a random error vector to a codeword of the corresponding error-correcting code $\mathcal{C}$ for its generator matrix while the Niederreiter Cryptosystem computes the syndrome vector for the parity check matrix.

These two cryptosystems explains how an error-correcting code is used in the corresponding one-way functions. Computing these functions involve matrix-vector multiplication and addition which can be done in polynomial-time.

McEliece Cryptosystem. We now describe the McEliece Encryption Scheme.

Secret-key. The underlying binary irreducible Goppa code is secret. The secret key is the triplet $(\mathbf{G}, \mathbf{P}, \mathbf{S})$ where $\mathbf{G}$ is the generator matrix of the underlying code. $\mathbf{P}$ is a $n \times n$ permutation matrix and $\mathbf{S}$ is a $k \times k$ non-singular matrix.

Public-key. The public-key is the matrix $\mathbf{S} \cdot \mathbf{G} \cdot \mathbf{P}$.

Encryption. For a randomly chosen $\mathbf{P}$ and $\mathbf{e}^\intercal \in \mathbb{F}_2^n$ a message $\mathbf{m}$ is encrypted as $\mathbf{y} = \mathbf{m} \cdot \mathbf{S} \cdot \mathbf{G} \cdot \mathbf{P} + \mathbf{e}$.

Decryption. $\mathbf{y}' = \mathbf{y} \cdot \mathbf{P^{-1}} = \mathbf{m} \cdot \mathbf{S} \cdot \mathbf{G} + \mathbf{e} \cdot \mathbf{P^{-1}}$. Here $\mathbf{m} \cdot \mathbf{S} \cdot \mathbf{G}$ is a valid codeword of the secret code and the permuted error vector $\mathbf{e} \cdot \mathbf{P}^{-1}$ has weight t. So it is possible for the decoder to correct these t errors. Patterson's algorithm is used to retrieve $m \cdot \mathbf{S}$ and thereby m is obtained.

Niederreiter Cryptosystem. This is the dual version of McEliece's cryptosystem. This too uses a permutation matrix of dimension $n \times n$ but it uses a non-singular matrix of dimension $(n-k) \times (n-k)$. The difference also lies in the choice of its secret key.

Secret key. The underlying code is secret. The parity-check matrix $\mathbf{H_{parity}}$ of the underlying code, $\mathbf{S}$ which is a non-singular matrix of dimension $(n-k) \times (n-k)$ and $\mathbf{P}$ which is a permutation matrix of dimension $n \times n$.

Public key. The matrix $\mathbf{S} \cdot \mathbf{H_{parity}} \cdot \mathbf{P}$.

Encryption. $\mathbf{y} = \mathbf{S} \cdot \mathbf{H_{parity}} \cdot \mathbf{P} \cdot \mathbf{m}$

Decryption. $\mathbf{S}^{-1} \cdot \mathbf{y} = \mathbf{H_{parity}} \cdot \mathbf{P} \cdot \mathbf{m}$.
$\mathbf{P} \cdot \mathbf{m}$ is retrieved by decoding from which $\mathbf{m}$ is obtained.

2.4 Information Set Decoding

Information Set Decoding algorithms obtain an error vector $\mathbf{e}$ with $|\mathbf{e}| = \omega$ when $\mathbf{H_0}$, $\mathbf{s}$, n, k and ω are given. These algorithms are heuristic in nature and take exponential time in terms of the code length to return a solution.

Information Set. *Information Set* (denoted as $\mathcal{J}$) is a set of random positions which are assumed to be free of errors such that $[\mathbf{H}'|\mathbf{I}_{n-k}]$ is a systematic form of $\mathbf{H_0}$, where $\mathbf{H}'$ is of dimension $(n-k) \times k$. The aim of ISD algorithms is to solve the *Computational Syndrome Decoding problem.*

Basic Steps of an ISD Algorithm Information Set Decoding algorithms have been the best known attack on Code-based encryption schemes. Let the size of the information set be k. The general steps of any ISD algorithm have been described below.

1. Choose an information set $\mathcal{J}$ with $\#\mathcal{J}$ being k and bring the parity-check matrix to the systematic form. If it is not possible to get this form then we must choose another information set.

2. Choose a set $\mathcal{I}$ of $p \geq 0$ points in the information set which are allowed to have errors.

3. Choose $\ell \geq 0$ points from the set $\{1, \ldots, n-k\}$.

4. Enumerate all the possible $\binom{k}{p}$ combinations.

5. For each combination a matrix-vector multiplication $\mathbf{H}' \cdot \mathbf{e}^\intercal$ is done. To perform this operation add the corresponding columns of $\mathbf{H}'$ such that $\mathbf{x} = \sum_{i \in \mathcal{I}} \mathbf{H}'_i$ by equation2.3.

6. Check the ℓ positions of the sum vector $\mathbf{x}$ with that of the $\mathbf{s}$ i.e., $\mathbf{s}_{[\ell]}$.

7. If a collision is obtained check whether the weight of $\mathbf{v} = \mathbf{x} + \mathbf{s}$ is $\omega - p$. The error vector is constructed with p errors in the positions corresponding to the p one-positions in the information set and the $\omega - p$ ones come from outside the information set according to the position of the ones in the sum vector. $\mathbf{e} = (e_1, \ldots, e_n)$ such that $e_i = 1$ if $i \in \mathcal{I}$ or $i \in [k+1, \ldots, n-k]$ and $\mathbf{v}[i] = 1$.

8. Otherwise go to step 1 and reiterate the entire process until success is obtained.

Algorithm 1 Basic Structure of ISD algorithm

Input: $\mathbf{H_0}$, $\mathbf{s_0}$, n, k and ω
Output: $\mathbf{e}$

1: **while** true **do**
2: Choose an information set $\mathcal{J}$ with $\#\mathcal{J}$ being k.
3: $\mathbf{H} \leftarrow \mathsf{LinAlg}(\mathbf{H_0})$. ▷ The subroutine LinAlg brings the parity-check matrix to the systematic or semi-systematic form. If it is not possible to get this form then we must randomly choose another information set.
4: Choose $p \geq 0$ points in the information set which are allowed to have errors.
5: $(\mathsf{flg}, \mathbf{e}) \leftarrow \mathsf{Search}_{\mathsf{error}}(\mathbf{H}, \mathbf{s}, \omega, p, \ell)$
6: **if** $\mathsf{flg} = 1$ **then return e**
7: **end if**
8: **end while**

Determining Time Complexity. Here p and ℓ are parameters of the ISD algorithm and they assume values from different ranges depending on the ISD algorithm being used. Let the success probability per iteration be denoted by π. A detailed discussion on the range of values of the parameters can be found in [23].

The expression of success probability per iteration π depends largely on the method of computing steps 4, 5, 6 and 7. As the computation varies with different ISD algorithms, the searching method of step 5 may be multi-level. For example, it is single-level in Leon [77], Lee-Brickell [76], Stern [101] and Dumer [46] while multi-level in MMT [78] and BJMM [10].

Algorithm 2 Construction of error vector when $|\mathcal{I}| = p$

Input: $\mathbf{H}$, $\mathbf{s}$, ω, p, ℓ

Output: $\mathcal{I}'$

1: Construct $List = \{\mathcal{I}, \mathbf{x} : \mathbf{x} = \sum_{i \in \mathcal{I}} \mathbf{H}_i,\ \#\mathcal{I} = p\}$ ▷ Enumerate all the possible $\binom{k}{p}$ combinations. For each combination a matrix-vector multiplication $\mathbf{H}' \cdot e^\intercal$ is done. To perform this operation add the corresponding columns of $\mathbf{H}'$ such that $\mathbf{x} = \sum_{i \in \mathcal{I}} \mathbf{H}'_i$ by equation 2.3.
2: **for all** $(\mathcal{I}, \mathbf{x})$ **do**
3: **if** $\mathbf{x}_{[\ell]} = \mathbf{s}_{[\ell]}$ **then** ▷ Check the ℓ positions of the sum vector $\mathbf{x}$ with that of the $\mathbf{s}$ i.e., $\mathbf{s}_{[\ell]}$.
4: **if** $|\mathbf{x} + \mathbf{s}| = \omega - p$ **then** ▷ If a collision is obtained check whether the weight of $\mathbf{v} = \mathbf{x} + \mathbf{s}$ is $\omega - p$.
5: $\mathbf{e} = (e_1, \ldots, e_n)$ such that $e_i = 1$ if $i \in \mathcal{I}$ or $i \in [k+1, \ldots, n-k]$ and $\mathbf{v}[i] = 1$ ▷ The error vector is constructed with p errors in the positions corresponding to the p one-positions in the information set and the $\omega - p$ ones come from outside the information set according to the position of the ones in the sum vector.
6: flg $\leftarrow 1$
7: **end if**
8: **end if**
9: **end for**

In fact step 4 too varies with different algorithms. The expected number of bit operations per iteration depends on the algorithm being used but all the algorithms have two broader parts concerning bit operations. One is the bit operations for linear algebra and the other one is the cost of enumerating exponential number of combinations. We denote the first component by T_{LA} and the second one by T_{SR}.

If after each iteration the information set is chosen independently then the expected number of iterations required is $1/\pi$.

$$\text{Expected number of bit operation per iteration} = T_{LA} + T_{SR}$$

If the information sets are chosen independently for each iteration then

$$\text{time complexity} = \frac{T_{LA} + T_{SR}}{\pi}.$$

Determining Memory Complexity. The exact size of a baselist and thus the intermediate lists depends on the specific algorithm being used.

2.5 Some Relevant Topics/Tools

We discuss some of the other concepts and tools that are used for improving the exhaustive search technique of step 5.

0/1-Knapsack Problem. Given set $\mathcal{X}$ of N positive integers and an integer target W. The goal is to find a subset $\mathcal{X}' \subseteq \mathcal{X}$ such that $\sum_{x \in \mathcal{X}'} x = W$.

Meet-in-the-Middle The Meet-in-the-middle (MITM) technique was first proposed by Horowitz and Sahni [64] in 1974 to provide a new algorithm for the 0/1-knapsack problem. If there are N positive integers and a target W then this algorithm reported a solution in $O(2^{N/2})$ time. To find a solution they separate the elements in two sets and form all possible combinations of those $N/2$ elements in one half and checks them on-the-fly against all possible combinations of $N/2$ elements of the other half. Diffie-Hellman used a similar concept in [44] to attack the double encryption. Double encryption, as the name suggests, requires two invocations of the encryption routine. The keys for each call are chosen independently. The ciphertext c is $Enc_{k_2}(Enc_{k_1}(P))$. Each of the keys are 56-bit long. So the total key length is 112 bits. To attack the double encryption, plaintext P is chosen and for each of the 2^{56} possible values of k_1, the encryption routine is invoked only once and these 2^{56} intermediate ciphertexts are stored in a table T. Now a ciphertext c' is decrypted only once by all the 2^{56} values of k_2 by invoking $Dec_{k_2(c')}$ and these partially decrypted ciphertexts are checked against the table T to search for a collision. We need about 2^{56} operations to find a collision. This attack is thus called Meet-In-The-Middle attack.

2-list problem [107]. Two lists $\mathcal{L}_1,\ \mathcal{L}_2$ with elements drawn randomly from $\{0,1\}^N$ are given. A collision can be obtained by a simple join operation. A collision is obtained with non-trivial probability if $\#\mathcal{L}_1 \times \#\mathcal{L}_2 \gg 2^N$. If the size of the two lists are at least $2^{N/2}$ collision can be found between the two lists in $O(2^{N/2})$. The memory complexity is also $O(2^{N/2})$.

Chapter 3

A Brief Survey of Relevant Literature

In this chapter we give a brief overview on the evolution of ISD algorithms starting with the Pranges's attack to the advanced algorithms which use Meet-In-The-Middle [64] and *representation technique* [65] as tools. The overview is focused on finding binary low weight error vector. We also briefly describe the code-based candidates of fourth round of NIST PQC standardization competition. Apart from the said topics we also mention about few other works and concepts with the aim of giving the reader a panoramic view of the research on ISD algorithms.

In sections 3.1 and 3.2 a brief survey of the ISD algorithms and the code-based candidates of fourth round of NIST PQC standardization competition- Classic McEliece [19], BIKE [4] and HQC [83] have been presented.

There are a few variants of the syndrome decoding problem like regular-syndrome decoding [6], d-split syndrome decoding and the Decoding-One-Out-of-Many(DOOM) problem. The regular-syndrome decoding [6] assumes the error vector is to be divided into some number of blocks where each block contains only one error whereas the d-split syndrome decoding [54] assumes the error-vector to be divided into d number of blocks and each of the d blocks contain equal number of errors. Sendrier has proposed a variation of syndrome decoding problem named DOOM in [97]. It refers to the successful retrieval of the error vector for any one of the multiple, say, N number of syndrome vectors. The computational version of this problem has been named Computational Syndrome Decoding- Multi. This work proposes a variant of the Stern's algorithm to obtain a complexity gain of $\sqrt{N}$ if N instances are given. The work also gives bounds on the value of N to get this gain. This variant is used to attack the WAVE [39] signature scheme.

3.1 Information Set Decoding Algorithms

In Chapter 2 the broader parts and steps of an ISD algorithm have been described. Here we give an overview of the works done on ISD algorithms with the main focus given on low-weight binary error vectors. The aim of these algorithms have been to speed-up the exhaustive search step as explained in Section 2.4 of Chapter 2. Most of them use the concept of meet-in-the-middle as a tool to meet this aim. Some of them additionally use

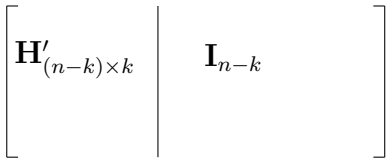


Figure 3.1: Systematic form of parity-check matrix for Prange's attack and Lee-Brickell attack

the *representation technique* proposed by Howgrave-Graham and Joux to solve an instance of the 0/1-knapsack problem. ISD algorithms work best for cases where the error-vector to be obtained is of low-weight. More specifically when ω is lesser than the Gilbert-Varshamov bound. Otherwise, the Generalized Birthday Algorithm performs better. Most of the ISD algorithms have been studied for $\omega/n \leq 0.5$. The other case is equivalent for the binary alphabet as shown in [27]. But this is not true for larger alphabets.

The work by Coffey and Goodman in [35] defines the complexity coefficient of ISD as $2^{n \cdot F(R)+o(n)}$ where $F(R) = \lim_{n\to\infty} \log_2 E(R)$ where $E(R)$ is the number operations.

There are three estimators [50, 52, 18] which calculate the time and memory requirement of several ISD algorithms.

Prange The information set of Prange's algorithm [91] has size k. By the definition of information set, these k positions are free of errors. Accordingly a permutation matrix $\mathbf{P}$ is applied on $\mathbf{H_0}$. A non-singular matrix $\mathbf{S}$ is applied on $\mathbf{H_0} \cdot \mathbf{P}$ and it is brought to the systematic form as in Figure 3.1. The value of the parameter p is 0. The positions corresponding to the columns of matrix $\mathbf{H'}$ are zero positions. The error vector to be obtained must have ω errors outside the information set. The size of the search space or the number of such error vectors to be tested corresponding to each permutation is $\binom{n-k}{\omega}$. If sum of the chosen columns of $\mathbf{I}_{n-k}$ and the syndrome vector $\mathbf{s}^\intercal$ is zero then a solution has been obtained. This equality must hold for all the $n-k$ bits i.e., for a set, say $\mathcal{Y}$, of ω positions $\{i_1, \cdots, i_\omega\} \subset \{k+1, \cdots, n\}$, $(\sum_{i\in\mathcal{Y}} \mathbf{H}_i + \mathbf{s}^\intercal)_{[n-k]} = \mathbf{0}^\intercal_{[n-k]}$. The expected number of iterations is $\binom{n}{\omega}/\binom{n-k}{\omega}$. This algorithm does not need to store any of the $\binom{n-k}{\omega}$ enumerations.

Lee-Brickell The original algorithm was proposed using generator matrix. Lee-Brickell [76] introduced the concept of allowing a small number of errors inside the information set but the size of the information set remained unchanged. The value of the parameter p is non-zero here. Among the total ω number of errors p errors comes from the columns of the matrix $\mathbf{H'}$. The rest of the $\omega - p$ errors come from the columns of $\mathbf{I}_{n-k}$. For each information set i.e., each iteration, $\binom{k}{p}$ combinations are enumerated from the information set. The columns corresponding to the one-positions in the information set are added to obtain the product of the $\mathbf{H'} \cdot \mathbf{e}^\intercal$ matrix vector multiplication and this sum is again added with the syndrome vector. If the sum has weight $\omega - p$ then the error vector is set to have ones in the p positions corresponding to the information set and the rest of the $\omega - p$ ones come from columns $\mathbf{I}_{n-k}$ such that the obtained sum vector had ones in those positions. The expected number of iterations is $\binom{n}{\omega}/(\binom{k}{p} \cdot \binom{n-k}{\omega-p})$. The $\binom{k}{p}$ enumerations are checked on-the-fly.

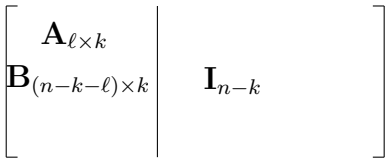


Figure 3.2: Systematic form of parity-check matrix for Leon's and Stern's attacks

Leon Leon's [77] attack is similar to that of Lee-Brickell with the exception being the introduction of early abortion of those combinations from the information set whose sum of columns of the one positions do not match with some $\ell < n-k$ number of positions corresponding to the syndrome vector. The size of the information set and size of the search space to be searched exhaustively due to the information set are same as that of Leon's. After permuting the columns of $\mathbf{H_0}$ and subsequently applying a non-singular matrix over $\mathbf{H_0}\cdot\mathbf{P}$ we obtain the matrix as in figure 3.2. This matrix is of same form as that in figure 3.1 but figure 3.2 helps to give an easy visualization of the new early abort technique and significance of the non-zero value of the parameter ℓ. For an information set $\mathcal{J}$ and the set of one-positions $\mathcal{I}$ the corresponding columns of $\mathbf{A}$ are added and if $\sum_{i\in\mathcal{I}}\mathbf{A}_i + \mathbf{s}^\intercal_{[\ell]}$ is not $\mathbf{0}^\intercal_{[\ell]}$ then this combination is discarded. Otherwise, check whether $\sum_{i\in\mathcal{I}}[\frac{\mathbf{A}}{\mathbf{B}}]_i + \mathbf{s}^\intercal$ has weight $\omega - p$. If yes, then a solution has been obtained and the rest of the ones come from $\mathbf{I}$ according to the positions of one in the sum vector of weight $\omega - p$. The expected number of iterations is $\binom{n}{\omega}/(\binom{k}{p}\cdot\binom{n-k-\ell}{\omega-p})$.

Stern Stern's attack [101] uses positive p and ℓ like the Leon's attack but it additionally introduces the use of the meet-in-the-middle approach to find a solution. The size of the information set $\mathcal{J}$ for Stern's attack is k. The information set is divided into two halves $\mathcal{J}_1$ and $\mathcal{J}_2$. Now each half has $p/2$ error positions. Two lists $\mathcal{L}_1$ and $\mathcal{L}_2$ are formed which stores the sum vectors corresponding to each of $\binom{k/2}{p/2}$ combinations from $[1, k/2]$ and $[k/2+1, k]$ respectively. The description of the lists may be seen as: $\mathcal{L}_j = \{\mathcal{I}_j, \sum_{i\in\mathcal{I}_j}\mathbf{H}'_i : \#\mathcal{I}_j = p/2 \text{ and } \mathcal{I}_j \subset \{(j-1)\cdot(k/2)+1, \ldots, (j-1)\cdot(k/2)+(k/2)\}\}$ where $j \in 1, 2$. Then following the meet-in-the-middle technique we check whether

$$\sum_{i\in\mathcal{I}_1}\mathbf{A}_i = \sum_{i\in\mathcal{I}_2}\mathbf{A}_i + \mathbf{s}^\intercal_{[\ell]}.$$

A collision is found with high probability if $\#\mathcal{L}_1 \cdot \#\mathcal{L}_2 \gg 2^\ell$. If a collision is obtained then we check whether $|\sum_{i\in\mathcal{I}_1}\mathbf{H}'_i + \sum_{i\in\mathcal{I}_2}\mathbf{H}'_i + \mathbf{s}^\intercal| = \omega - p$. If the sum vector has weight $\omega - p$ then an error vector is constructed with $p/2$ ones coming from each of the halves of the information set and the remaining ones come from outside the information set according to the position of the ones in the obtained sum vector. Otherwise we must restart with a new information set. In this algorithm the search space of size $\binom{k}{p}$ is not searched exhaustively. The information set is divided into two halves. A search space of size $\binom{k/2}{p/2}^2$ is searched. The expected number of iterations is $\binom{n}{\omega}/(\binom{k/2}{p/2}^2\cdot\binom{n-k-\ell}{\omega-p})$. The memory complexity is of order $O(\#\mathcal{L}_1)$. The second list $\mathcal{L}_2$ can be checked on-the-fly and the final list need not be stored.

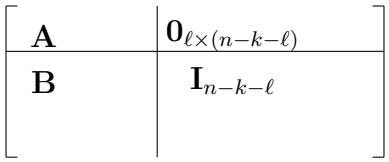


Figure 3.3: Matrix manipulation for Dumer's attack

All of the works which followed up this work solves the equation

$$\sum_{i\in\mathcal{I}_1}\mathbf{A}_i = \sum_{i\in\mathcal{I}_2}\mathbf{A}_i + \mathbf{s}^{\mathsf{T}}_{[\ell]}.$$

Each of them attempts to solve it in a lower time complexity than the preceding ones. To meet this aim some of them do multi-level collision search and use representation technique.

Dumer Though the Stern's attack has many advantages, it imposes one constraint on the structure of the error vector. The algorithm checks whether $\sum_{i\in\mathcal{I}_1}\mathbf{A}_i+\sum_{i\in\mathcal{I}_2}\mathbf{A}_i+s^{\mathsf{T}}_{[\ell]} = \mathbf{0}^{\mathsf{T}}_{[\ell]}$. If yes then it checks $|\sum_{i\in\mathcal{I}_1}\mathbf{H}'_i + \sum_{i\in\mathcal{I}_2}\mathbf{H}'_i + \mathbf{s}^{\mathsf{T}}| = \omega - p$. It can be observed that those ℓ places do not contribute to the weight of the vector. Thus there is a run of ℓ zeroes. Dumer [46] eliminated the necessity to have a run of ℓ zeroes in the error vector. The representation of the parity-check matrix for Dumer's algorithm is shown in Figure 3.3. Thus we gain some flexibility for distributing the ones in the error vector. This algorithm has a bigger information set of size $k+\ell$ which, like the Stern's algorithm, is divided into two halves. The size of the lists are $\binom{(k+\ell)/2}{p/2}$. For same $n,\ k,\ p$ and ℓ these lists are larger than those of Stern's. The expected number of iterations is $\binom{n}{\omega}/\binom{(k+\ell)/2}{p/2}^2\cdot\binom{n-k-\ell}{\omega-p}$.

Ball-Collision Like the Dumer's attack the ball-collision algorithm also eliminates the run of ℓ zeroes but it removes the run without increasing the size of the information set. Unlike the Dumer's algorithm the size of each baselist remains $\binom{k/2}{p/2}$. Additional q error positions are chosen from ℓ positions outside the information set apart from the remaining $\omega-p-q$ errors. The collision is searched at ℓ positions but it checks whether the sum vector has weight $\omega-p-q$ instead of checking for exact match at those ℓ positions.

$$\sum_{i\in\mathcal{I}_1}\mathbf{A}_i + \mathbf{x_0} = \sum_{i\in\mathcal{I}_2}\mathbf{A}_i + \mathbf{y_0} + \mathbf{s}^{\mathsf{T}}_{[\ell]},$$

where $\mathbf{x_0}\in\mathbb{F}_2^{\ell/2}$ and $\mathbf{y_0}\in\mathbb{F}_2^{\ell/2}$. The expected number of collisions is $\binom{n}{\omega}/(\binom{k/2}{p/2}^2\cdot\binom{\ell/2}{q/2}^2\cdot\binom{n-k-\ell}{\omega-p-q})$.

Representation Technique and MMT Representation technique for solving *0/1*-knapsack problem was introduced in Eurocrypt 2010 by Howgrave-Graham and Joux in [65]. This technique has been adopted in MMT [78] and BJMM [10]. We describe this technique first and then we move on to ISD algorithms again.

Similar to the meet-in-the-middle, two equal parts of the input set is formed and $2^{N/2}$ combinations are obtained from each of the parts. The exception lies in the way of selection of $N/2$ integers from the set of size N. Instead of dividing the set into two disjoint sets and enumerating all possible $2^{N/2}$ combinations from the selected half, the work in [65] chooses $N/2$ integer from the N integers. The proposed decomposition is as follows:

$$\sum_{i=1}^{N} a_i \cdot x_i = W + \sum_{i=1}^{N} a_i \cdot y_i,$$

where $x_i \in \{0,1\}$, $y_i \in \{0,1\}$ and $i \in \{1, \cdots, N\}$. Each of the two sub-knapsacks have $N/2$ integers. So each solution of the above equation has $\binom{N}{N/2}$ representations.

Figure 3.4 gives an inverted tree representation of this algorithm. Like Dumer's attack MMT also divides the information set into two halves of size $(k+\ell)/2$. This algorithm exploits the decompositions of one: $1 = 1 + 0,\ 1 = 0 + 1$ following the work in [65]. The original work in [78] using depth-2 tree requires four baselists. Each baselist has one entry corresponding to each of the $\binom{(k+\ell)/2}{p/4}$ combinations. Let us denote the i-th baselist as L_{1i} where $i \in \{1,\ 2,\ 3,\ 4\}$ and let $\mathcal{I}$ be the set of one positions from the information set $\mathcal{J}$ and $\#\mathcal{I} = p$. Let $\mathcal{I} = \mathcal{I}_1 \cup \mathcal{I}_2$ and $\#\mathcal{I}_1 = \#\mathcal{I}_2 = p/2$. For $j \in 1, 3$, $L_{1j} = \{\mathcal{I}_{1j}, \sum_{i\in\mathcal{I}} \mathbf{H}'_i :\ \mathcal{I}_{1j} \subset \{1, \ldots, (k+\ell)/2\}$ and $\#\mathcal{I}_{1j} = p/4\}$ and for $j \in 2, 4$, $\mathcal{L}_{1j} = \{\mathcal{I}_{1j}, \sum_{i\in\mathcal{I}} \mathbf{H}'_i :\ \mathcal{I}_{1j} \subset \{(k+\ell)/2 + 1, \ldots, (k+\ell)\}$ and $\#\mathcal{I}_{1j} = p/4\}$. Following the meet-in-the-middle technique, the second components of $\mathcal{L}_{11}$ and $\mathcal{L}_{12}$ are added mod 2 and then filtered based on ℓ_1 positions. The new tuples are stored in $\mathcal{L}_1$. Similarly we obtain $\mathcal{L}_2$ from $\mathcal{L}_{13}$ and $\mathcal{L}_{14}$. The expected number of tuples in $\mathcal{L}_1 = \max\{\#\mathcal{L}_{11}, (\#\mathcal{L}_{11} \cdot \#\mathcal{L}_{12})/2^{\ell_1}\}$ and $\mathcal{L}_2 = \max\{\#\mathcal{L}_{13}, (\#\mathcal{L}_{13} \cdot \#\mathcal{L}_{14})/2^{\ell_1}\}$. Unlike Stern or Dumer, for each of $\mathcal{L}_1$ and $\mathcal{L}_2$ MMT chooses $p/4$ one-positions from one half and the another $p/4$ one-positions from the other half of the information set. Thus finally one solution has $\binom{p/2}{p/4}^2$ representations. The partial error vectors corresponding to these second level lists forms a subset of all possible vectors having $(k+\ell)/2$ length vectors with weight $p/2$. The last merge of $\mathcal{L}_1$ and $\mathcal{L}_2$ gives us partial error vectors with $k+\ell$ length and p weight.

BJMM This attack, like the MMT algorithm, exploits representation technique but this additionally considers decomposition of zero: 0=1+1. The size of the information set remains unchanged but the number of ones per information set is slightly larger. The two final information sets of size, $\mathcal{J}_1$ and $\mathcal{J}_2$ are of size $(k+\ell)/2$ each with having $p/2+\epsilon$ ones. These extra ϵ ones must come from the same ϵ positions. Due to these extra ϵ positions the number of representations of a solution is $\binom{p/2}{p/4}^2 \cdot \binom{k+\ell-p/2}{\epsilon}^2$. The original algorithm was presented as a depth-3 inverted tree structure.

Esser et al. has come up with the first concrete implementations of the ISD algorithms in MMT [78] and BJMM [10] in [51].

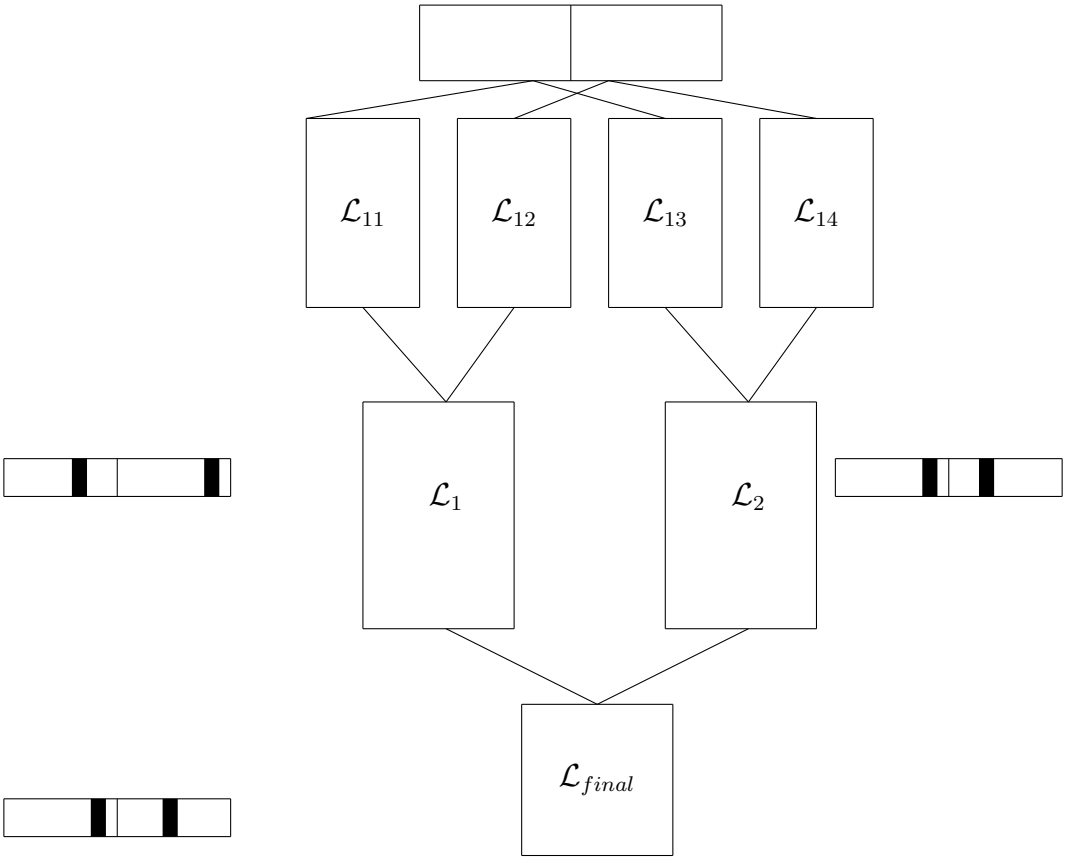
Statistical decoding 2.0 This work in [31] follows a different approach than the ISD algorithms mentioned so far. It does not do partial Gaussian elimination and its information set is of size, say, $s < n$. The matrix vector multiplication $(\mathbf{H_0} \cdot e)$ is interpreted as

$$\mathcal{L}_{11} = \{\mathcal{I}_{11}, \sum_{i\in\mathcal{I}_{11}} \mathbf{H}'_i, \text{ where } \mathcal{I}_{11} \subset \{1, \ldots, (k+\ell)/2\} \text{ and } \#\mathcal{I}_{11} = p/4\}$$

$$\mathcal{L}_{12} = \{\mathcal{I}_{12}, \sum_{i\in\mathcal{I}_{12}} \mathbf{H}'_i, \text{ where } \mathcal{I}_{12} \subset \{(k+\ell)/2+1, \ldots, (k+\ell)\} \text{ and } \#\mathcal{I}_{12} = p/4\}$$

$$\mathcal{L}_{13} = \{\mathcal{I}_{13}, \sum_{i\in\mathcal{I}_{13}} \mathbf{H}'_i, \text{ where } \mathcal{I}_{13} \subset \{1, \ldots, (k+\ell)/2\} \text{ and } \#\mathcal{I}_{13} = p/4\}$$

$$\mathcal{L}_{14} = \{\mathcal{I}_{14}, \sum_{i\in\mathcal{I}_{14}} \mathbf{H}'_i, \text{ where } \mathcal{I}_{14} \subset \{(k+\ell)/2+1, \ldots, (k+\ell)\} \text{ and } \#\mathcal{I}_{14} = p/4\}$$



Original MMT

Figure 3.4: Inverted-tree like representation of the original MMT algorithm

LPN problem. The first s columns of $\mathbf{H_0}$ are chosen for a linear combination. The linear combination of the remaining $n-s$ columns form the noise of LPN problem. Fast Fourier Transform is used to recover the first s bits of the error vector. To retrieve the error vector from the second linear combination the BJMM algorithm is used.

May and Ozerov have proposed an ISD algorithm in [79] which is asymptotically superior to the ISD algorithms proposed by Prange, Stern, Dumer, May et al., Both et al. but it has a polynomial overhead due to the use of a recursion tree. This overhead makes the algorithm impractical [50]. Both and May have proposed another algorithm based on [79] in [26].

Time/Memory Trade-offs

Time/Memory trade-offs of ISD algorithms have been studied in [53, 67, 23]. The work in [53] proposed a trade-off of the MMT [78] and BJMM [10] using the Schroeppel-Shamir algorithm [96, 65]. This trade-off has been designed for binary codes considering hamming metric. The Schroeppel-Shamir algorithm is used to reduce the size of the baselists. The sub-matrix matching problem is solved through two levels in MMT. This work replaces the meet-in-the-middle routines with Schroeppel-Shamir algorithm. For same size of input lists Schroeppel-Shamir algorithm has same time-complexity but a reduced memory complexity. With respect to the memory requirement M of the original MMT algorithm this new version claims about a memory requirement of about $M^{1/4}$.

The work in [27] proposes an ISD algorithm focused on syndrome decoding with large ternary weight. This too tries to solve submatrix matching problem but it works on $\mathbb{F}_q^n$ instead of $\mathbb{F}_2^n$ where $q > 2$. This work uses Wagner's method of solving generalized birthday problem [107]. It also proposes a smoothed version of Wagner's algorithm. Later it studies the application of representation technique together with Wagner's method to solve this version of sub-matrix matching problem and proposes partial representation which gives better performance for this version. The application of this new ISD algorithm is studied on the WAVE signature scheme which is based on hardness of syndrome decoding problem over $\mathbb{F}_3$.

The work in [67] gives a time/memory trade-off for ISD algorithm proposed in the previous work [27]. It uses the technique of dissection [45] to replace the Wagner's algorithm to solve the sub-matrix matching problem and proposes a new algorithm based on [27] with lesser memory requirement.

Another method of choosing Information Sets

In Chapter 2 we have mentioned that the information set is chosen independently in each iteration. Canteaut and Chabaud in [28] has proposed a different way of choosing information sets per iteration. Detailed analysis of this new approach was done by Canteaut and Sendrier [29]. The original work has considered Stern's algorithm for obtaining the weight-p error vector. The difference is that the successive information sets are not independent. The new algorithm has to be modelized as a discrete time stochastic process. They have further argued that this discrete time stochastic process is a homogeneous Markov chain. This new approach improves the performance of the Stern's attack.

Instead of choosing a random information set for every iteration they have proposed to obtain a new information set from the existing one by replacing only one position of the existing information set. If the existing information set is denoted by $\mathcal{J}$ then the next information set $\mathcal{J}'$ may be chosen by following Algorithm 3.

Algorithm 3 Choosing new information set using the algorithm in [29]

Input: $\mathbf{H}$, $\mathcal{J}$
Output: $\mathcal{J}'$

1: **while** true **do**
2: Choose randomly a column $i \in \mathcal{J}$ and another column $j \in \{\{1, \cdots, k\} \setminus \mathcal{J}\}$
3: **if** $\mathbf{H}'_{i,j} = 1$ **then** $\mathcal{J}' = \{\mathcal{J} \setminus \{i\}\} \cup \{j\}$ **return** $\mathcal{J}'$
4: **end if**
5: **end while**

3.2 Some of the Code-based Candidates of NIST PQC Standardization Competition

The submissions in code-based category in NIST PQC standardization competition are based on either of the following distance metrics- hamming and rank. All the four candidates in the fourth round [2] are based on hamming metric. The schemes based on rank metric considered till the second round were ROLLO [81] and RQC [82]. Before moving on to the fourth-round candidates we briefly define the rank metric.

The rank weight of a vector $\mathbf{v}$ is the rank of an associated matrix which is defined by the map $F_i : GF(q^m) \mapsto GF(q)$. The rank distance between two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as the rank weight of the vector $\mathbf{x} - \mathbf{y}$ which in turn is equal to the rank of a matrix M whose elements m_{ij} are defined as $F_i(v_j)$. ROLLO and RQC were considered up to round 2. During round 1 attack algorithms against Rank Syndrome Decoding [58] were not well studied. Later algebraic attacks were proposed. As a consequence new parameters were proposed for both RQC and ROLLO. This lead to increase in key and ciphertext sizes compared to the previous parameter set. The NIST did not select these two schemes for the next round stating that the security analysis of RQC and ROLLO need more time to mature[1].

Classic McEliece This is a KEM which has been designed following the original McEliece Cryptosystem [80] but in the Niederreiter dual version. The underlying error-correcting code is a binary irreducible Goppa code. The original algorithm is OW-CPA secured and can be converted to CCA-2 secured version [69] using Fujisaki-Okamoto transformation. Classic McEliece is secured against IND-CCA-2 attacks in random oracle model at security level high enough to be secured against attacks by quantum adversaries. The most powerful cryptanalysis of this KEM is due to ISD. The ciphertext size is shortest among that of all KEMs and there is no decoding failure due to the Goppa code. The downsides are its large public key size and slow key-generation algorithm. Since it is a KEM, major constituent

algorithms are the key generation algorithm, encapsulation algorithm and decapsulation algorithm. The encapsulation algorithm takes the parity check matrix and random error vector as inputs to produce the syndrome vector as the ciphertext. Then the session key is obtained using the SHAKE256 algorithm using the syndrome-vector and error vector. The decapsulation algorithm decodes the ciphertext to retrieve the error vector and recomputes the session key using SHAKE256.

BIKE This is another KEM candidate in the fourth round of NIST PQC standardization competition. Like the Classic McEliece, its underlying PKE also follows the Niederreiter-style but the error-correcting code is Quasi-Cyclic Moderate Density Parity Check (QC-MDPC) code. A circulant matrix is a square matrix whose each row is a right shift of its previous row. Thus such a matrix can be defined by its first row only. A block circulant matrix is a square matrix which is composed of smaller circulant sub-matrices. The number of such blocks is its index and number of elements per row of such a block is its order. The parity-check matrix of QC-MDPC code is a block circulant matrix with around $\sqrt{n}$ number of ones per row. There is an isomorphism between $r \times r$ binary circulant matrices and the quotient polynomial ring $\mathbb{F}_2[x]/(x^r - 1)$. So any operations on such matrices involve operations with polynomials. The secret key is a $\mathbf{H}_{\mathbf{0}_{r \times 2 \cdot r}}$ binary matrix where r is a prime and the public key is the systematic form of the secret key. Hence the size of the public key is not large. IND-CPA security of the underlying PKE in the random oracle model depends on the difficulty of solving the decisional Quasi-cyclic Syndrome Decoding (QCSD) and the decisional Quasi-cyclic Codeword Finding (QCCF) problems and Fujisaki-Okamato transform is used to obtain the IND-CCA secured version.

HQC The security of this KEM too depends on the hardness of decoding random binary QC-MDPC codes. This scheme uses two codes: a $\mathcal{C}_{n,k}$ which can correct up to d errors and a random double-circulant $[2 \cdot n, n]$ code. The first code is instantiated by concatenated Reed-Muller and Reed-Solomon codes. The latter code is used to generate some error which can be recovered by $\mathcal{C}$. These codes are not used as hidden trapdoors unlike the other two schemes. The secret key is a tuple (x, y) drawn randomly from $\mathcal{R}_\omega^2$ where $\mathcal{R}_\omega = \{\mathbf{v} : \mathbf{v} \in \mathcal{R},\ |\mathbf{v}| = \omega\}$ and $\mathcal{R} = \mathbb{F}[x]/(x^n - 1)$. The public key is another tuple $(h, s = x + h \cdot y)$ where h is drawn randomly from $\mathcal{R}$. This is IND-CPA secured and can be transformed into IND-CCA secured using Fujisaki-Okamoto transformation. The cipher text $c = (u, v) = (r_1 + h \cdot r_2, m \cdot G + s \cdot r_2 + e)$. To decrypt, the receiver uses the decoding algorithm to decode $(v - u \cdot y)$.

Chapter 4

Generalised Stern's ISD Algorithm

In this chapter we introduce a generalisation of Stern's ISD algorithm [101]. Various prominent ISD algorithms have been described in Chapter 3. Among them the algorithm by Stern is a watershed in the literature of ISD algorithms. While the later works have led to more faster algorithms, the community still continues to use Stern's algorithm as the baseline ISD algorithm to evaluate code based cryptosystems. This is exemplified by the discussion on the PQC forum available at [57].

Stern's algorithm as well as its variant due to Dumer [46] (and later by Finiasz and Sendrier) are obtained as special cases of the generalisation proposed in this chapter. Further, a recent generalisation of Stern's algorithm due to Bernstein and Chou [18] is also obtained as a special case. We also briefly sketch how our generalisation can be further extended to obtain a unified algorithm from which Prange's [91], Lee and Brickell's [76], Leon's [77] as well as Stern's [101], Dumer's [46], and Bernstein and Chou's [18] algorithms are obtained as special cases.

Dumer's algorithm gives a novel technique which allows to choose the ones for the information set from an increased number of positions. Hence for same $n,\ k,\ p$ and ℓ the number of iterations are smaller than that of Stern's but this also leads to increased size of the information set and therefore increased list sizes with respect to Stern's algorithm. The exponents in the asymptotic worst-case complexities of Stern's attack and Dumer's attack are $0.05563n$ and $0.05558n$ respectively.

The generalisation of Stern's algorithm that has been proposed in this chapter is due to the introduction of two new parameters λ and δ. The parameter λ takes integer values in the range $[-\ell, \ell]$ and $\delta \in (0, 1]$. The introduction of the parameter λ helps to generalise the Stern's algorithm such that when $\lambda = 0$ the generalised algorithm is same as Stern's algorithm and for $\lambda = \ell$ the generalised version is Dumer's algorithm.

This work is based on the first contribution of the work in [23].

4.1 Notation

For a subset S of $[k]$ by $\chi(S)$ we will denote the k-bit string $(x_1, \ldots, x_k)$ such that $x_j = 1$ if and only if $j \in S$. Otherwise $x_j = 0$.

4.2 A generalisation of Stern's ISD algorithm

We now describe the proposed generalisation of Stern's algorithm. The generalisation is obtained through the use of two parameters λ and δ in addition to the parameters ℓ and p used in Stern's algorithm. The maximum possible ranges of values of the parameters are as follows.

$$0 \leq \ell \leq r, \qquad -(k-2) \leq \lambda \leq \ell, \qquad 0 \leq p \leq \min(k+\lambda, \omega), \qquad \delta \in (0, 1]. \tag{4.1}$$

Remark 1. *A generalisation of Stern's algorithm (called* isd1*) has been given by Bernstein and Chou [18, Section 4.8] through the use of a parameter z, which is related to the parameter λ that we use by $z + \lambda = \ell$. The only values of z mentioned in [18] are $z = 0$ and $z = \ell$. It is not clear whether [18] allows z to be greater than ℓ (corresponding to λ less than 0). Note that due to the use of the additional parameter δ, our generalisation of Stern's algorithm subsumes the generalisation in [18]. As we will see later (Remark 7 in Section 4.3), it is the parameter δ that turns out to determine the non-triviality of the generalisation that we propose.*

The basic structure of the algorithm is shown in Algorithm 4. It consists of a linear algebra step and a search step as in Stern's algorithm. We shall now describe these two steps. The input to the algorithm is $(\mathbf{H}_0, \mathbf{s}_0, \omega)$ and the requirement is to find a vector $\mathbf{e} \in \mathbb{F}_2^n$ such that $\mathbf{H}_0\mathbf{e}^\top = \mathbf{s}_0^\top$.

Algorithm 4 A general formulation of Stern's ISD algorithm.

Input: $(\mathbf{H}_0, \mathbf{s}_0, \omega)$
Output: $\mathbf{e}$ such that $\mathbf{H}_0\mathbf{e}^\top = \mathbf{s}_0^\top$ and $\mathsf{wt}(\mathbf{e}) = \omega$.

1: **while** true **do**
2: Choose a random permutation matrix $\mathbf{P}$ of order $n \times n$
3: $(\mathbf{A}_1, \mathbf{A}_2, \mathbf{B}_1, \mathbf{B}_2, \mathbf{u}, \mathbf{v}) \leftarrow \mathsf{LinAlg}(\mathbf{H}_0, \mathbf{P}, \mathbf{s}_0, \ell, \lambda)$
4: $(\mathsf{flg}, \mathbf{e}) \leftarrow \mathsf{Srch}(\mathbf{A}_1, \mathbf{A}_2, \mathbf{B}_1, \mathbf{B}_2, \mathbf{u}, \mathbf{v}, \omega, p, \delta)$
5: **if** $\mathsf{flg} = \mathsf{yes}$ **then return** $\mathbf{Pe}^\top$
6: **end if**
7: **end while**

Linear algebra. Apply Gaussian elimination using row operations to the augmented matrix $[\mathbf{H}_0\mathbf{P} \,|\, \mathbf{s}_0^\top]$ to obtain a matrix $[\mathbf{H} \,|\, \mathbf{s}^\top]$, where $\mathbf{H} = \mathbf{U}\mathbf{H}_0\mathbf{P}$ and $\mathbf{s}^\top = \mathbf{U}\mathbf{s}^\top$ are of the

following forms

$$\mathbf{H} = \begin{array}{|c|c|c|} \hline \mathbf{A}_{\ell\times(k+\lambda)} & \mathbf{C}_{\ell\times(\ell-\lambda)} & \mathbf{0}_{\ell\times(r-\ell)} \\ \hline \mathbf{B}_{(r-\ell)\times(k+\lambda)} & \mathbf{D}_{(r-\ell)\times(\ell-\lambda)} & \mathbf{I}_{r-\ell} \\ \hline \end{array}, \qquad \mathbf{s}^\top = \begin{array}{|c|} \hline \mathbf{u}^\top \\ \hline \mathbf{v}^\top \\ \hline \end{array} \tag{4.2}$$

Here $\mathbf{U}$ is the invertible matrix of order $r \times r$ which corresponds to the sequence of row operations, $\mathbf{u} \in \mathbb{F}_2^\ell$ and $\mathbf{v} \in \mathbb{F}_2^{r-\ell}$.

Remark 2. *Consider the event E_1 that $\mathbf{H}_0\mathbf{P}$ can be reduced to the form shown in (4.2), i.e. the bottom right sub-matrix of $\mathbf{H}_0\mathbf{P}$ of order $(r-\ell)\times(r-\ell)$ is invertible. So E_1 is the event that the linear algebra step succeeds, and let π_1 be the probability of E_1. Under the heuristic assumption that the entries of $\mathbf{H}_0$ are independent and uniform random bits, $\pi_1 = \prod_{i=1}^{r-\ell}(1-2^{-i})$, which is about 0.288 for large enough values of $r-\ell$.*

Write

$$\mathbf{A} = [\mathbf{A}_1|\mathbf{A}_2] \quad \text{and} \quad \mathbf{B} = [\mathbf{B}_1|\mathbf{B}_2], \tag{4.3}$$

where $\mathbf{A}_1$ (resp. $\mathbf{A}_2$) is an $\ell \times \lfloor (k+\lambda)/2 \rfloor$ (resp. $\ell \times \lceil (k+\lambda)/2 \rceil$) matrix, and $\mathbf{B}_1$ (resp. $\mathbf{B}_2$) is an $(r-\ell)\times\lfloor (k+\lambda)/2\rfloor$ (resp. $(r-\ell)\times\lceil (k+\lambda)/2\rceil$) matrix. The call $\mathsf{LinAlg}(\mathbf{H}_0, \mathbf{P}, \mathbf{s}_0, \ell, \lambda)$ in Algorithm 4 returns $(\mathbf{A}_1, \mathbf{A}_2, \mathbf{B}_1, \mathbf{B}_2, \mathbf{u}, \mathbf{v})$.

Search. As explained in 3.1, it is sufficient to obtain a vector $\mathbf{e} \in \mathbb{F}_2^n$ such that $\mathbf{H}\mathbf{e}^\top = \mathbf{s}^\top$. The search step looks for such an $\mathbf{e}$. The parameter δ is used in the search step. As in Stern's algorithm, the search step is done in two parts. The first part prepares a list $\mathcal{L}$ and the second part searches for a collision. The difference with Stern's algorithm is that instead of storing all the $\binom{\lfloor (k+\lambda)/2 \rfloor}{\lfloor p/2 \rfloor}$ ℓ-bit strings arising from considering all $\lfloor p/2 \rfloor$ possible combinations of the $\lfloor (k+\lambda)/2 \rfloor$ columns of $\mathbf{A}_1$, only $\binom{\lfloor (k+\lambda)/2 \rfloor}{\lfloor p/2 \rfloor}^\delta$ of such combinations are considered and the corresponding ℓ-bit strings stored in $\mathcal{L}$. The second part of the search step proceeds more or less in the same manner as that of Stern's algorithm. The complete search algorithm is shown in Algorithm 5. Note that the list $\mathcal{L}$ is stored indexed on the first component of its entries.

Remark 3. *From the description of the algorithm, we note that for the error vector $\mathbf{e}$ returned by the algorithm, the matrix vector product $\mathbf{H}\mathbf{e}^\top$ is of the following form.*

$$\begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{C} & \mathbf{0}_{\ell\times(r-\ell)} \\ \mathbf{B}_1 & \mathbf{B}_2 & \mathbf{D} & \mathbf{I}_{(r-\ell)} \end{bmatrix} \begin{bmatrix} \mathbf{e}_1^\top \\ \mathbf{e}_2^\top \\ \mathbf{0}_{\ell-\lambda}^\top \\ \mathbf{e}_3^\top \end{bmatrix}. \tag{4.4}$$

Note that there are $\ell - \lambda$ positions where $\mathbf{e}$ has the value 0.

Algorithm 5 The Srch procedure.

Input: $(\mathbf{A}_1, \mathbf{A}_2, \mathbf{B}_1, \mathbf{B}_2, \mathbf{u}, \mathbf{v}, \omega, p, \delta)$(*see* (4.2)*and* (4.3))
Output: $(\mathsf{yes}, \mathbf{e})$ such that $\mathbf{He}^\top = \mathbf{s}^\top$ and $\mathsf{wt}(\mathbf{e}) = \omega$, or $(\mathsf{no}, \perp)$.

$c \leftarrow k + \lambda$; $L_1 \leftarrow \binom{\lfloor c/2 \rfloor}{\lfloor p/2 \rfloor}$
$\mathcal{L} \leftarrow ()$; $i = 0$
for $S \subseteq [\lfloor c/2 \rfloor]$ with $\#S = \lfloor p/2 \rfloor$ **do**
    if $i < \lceil L_1^\delta \rceil$ **then**
        $\mathbf{e}_1 = \chi(S)$; $\mathbf{a}_1^\top = \mathbf{A}_1 \mathbf{e}_1^\top$;
        $\mathbf{b}_1^\top = \mathbf{B}_1 \mathbf{e}_1^\top$
        $\mathcal{L} \leftarrow \mathcal{L} \cup \{(\mathbf{a}_1, \mathbf{b}_1, S)\}$
        $i \leftarrow i + 1$
    else
        break
    end if
end for
for $T \subseteq [\lceil c/2 \rceil]$ with $\#T = \lceil p/2 \rceil$ **do**
    $\mathbf{e}_2 = \chi(T)$; $\mathbf{a}_2^\top = \mathbf{A}_2 \mathbf{e}_2^\top$;
    $\mathbf{b}_2^\top = \mathbf{B}_2 \mathbf{e}_2^\top$
    for all $(\mathbf{a}_1, \mathbf{b}_1, S) \in \mathcal{L}$ such that $\mathbf{a}_1 + \mathbf{u} = \mathbf{a}_2$ **do**
        if $\mathsf{wt}(\mathbf{b}_1 + \mathbf{b}_2 + \mathbf{v}) = \omega - p$ **then**
            $\mathbf{e}_1 = \chi(S)$; $\mathbf{e}_3 = \mathbf{b}_1 + \mathbf{b}_2 + \mathbf{v}$ **return** $(\mathsf{yes}, [\mathbf{e}_1 | \mathbf{e}_2 | \mathbf{0}_{\ell - \lambda} | \mathbf{e}_3])$
        end if
    end for
end for
    return $(\mathsf{no}, \perp)$

Correctness. It is easy to check that any solution returned by Algorithm 4 is correct.

Special cases. If we take $\delta = 1$, then we obtain the generalisation of Stern's algorithm by Bernstein and Chou [18] (see Remark 1). If we take $\delta = 1$ and $\lambda = 0$, then we essentially get back Stern's algorithm, and if we take $\delta = 1$ and $\lambda = \ell$, then we essentially obtain Dumer's algorithm.

Further generalisation. The matrix $\mathbf{A}$ has been divided into $\mathbf{A}_1$ and $\mathbf{A}_2$, where $\mathbf{A}_1$ (resp. $\mathbf{A}_2$) has $\lfloor (k + \lambda)/2 \rfloor$ (resp. $\lceil (k + \lambda)/2 \rceil$) columns. We may instead let $\mathbf{A}_1$ and $\mathbf{A}_2$ to have κ_1 and κ_2 columns respectively, where κ_1 and κ_2 are two parameters which are non-negative integers satisfying $\kappa_1 + \kappa_2 = k + \lambda$. Correspondingly, we divide $\mathbf{B}$ into matrices $\mathbf{B}_1$ and $\mathbf{B}_2$ having κ_1 and κ_2 columns respectively. Let p_1 and p_2 be two additional parameters which are non-negative integers such that $p_1 + p_2 = p$.

The generalisation is the following. The first part of the search step will consider column combinations of $\mathbf{A}_1$ taken p_1 at a time and the second part will consider column combinations of $\mathbf{A}_2$ taken p_2 at a time. The description of the search step given in Algorithm 5 can be easily modified to give effect to this generalisation: replace $\lfloor c/2 \rfloor$ by κ_1, $\lceil c/2 \rceil$ by κ_2, $\lfloor p/2 \rfloor$ by p_1 and $\lceil p/2 \rceil$ by p_2.

With the above generalisation, we obtain a *unified* algorithm which provides as special cases all the algorithms from Prange's to Dumer's. It is clear that the generalised Stern's algorithm is obtained by taking $\kappa_1 = \lfloor (k+\lambda)/2 \rfloor$, $\kappa_2 = \lceil (k+\lambda)/2 \rceil$, $p_1 = \lfloor p/2 \rfloor$ and $p_2 = \lceil p/2 \rceil$. Prange's algorithm is obtained by setting $\ell = \lambda = \kappa_1 = p = p_1 = p_2 = 0$, $\delta = 1$ and $\kappa_2 = k$; Lee and Brickell's algorithm is obtained by setting $\ell = \lambda = \kappa_1 = p_1 = 0$, $\delta = 1$, $\kappa_2 = k$ and $p_2 = p$; and Leon's algorithm is obtained by setting $\lambda = \kappa_1 = p_1 = 0$, $\delta = 1$, $\kappa_2 = k$ and $p_2 = p$.

Practical efficiency improvements. There are known techniques for improving the practical efficiency of Stern's algorithms. Below we briefly describe two of these techniques which we will use in estimating the expected number of bit operations.

Chase's sequence: For each choice S, the Srch procedure obtains $\mathbf{e}_1 = \chi(S)$ and computes $\mathbf{a}_1^\top = \mathbf{A}_1 \mathbf{e}_1^\top$. The latter computation involves adding together $\lfloor p/2 \rfloor$ columns of $\mathbf{A}_1$. This requires a total of $\lfloor p/2 \rfloor - 1$ additions of ℓ-bit vectors. Since L_1 subsets S are considered, the total number of ℓ-bit vector additions is $L_1 \cdot (\lfloor p/2 \rfloor - 1)$. An alternative way to perform the entire computation is to use Chase's sequence (see Section 7.2.1.3 of [68]) as was done in [105, 108]. In this technique, the subsets S are generated incrementally, where the next subset is obtained from the present subset by removing one element and including a new one. So from the vector $\mathbf{a}_1$ corresponding to the present subset, the vector $\mathbf{a}_1$ corresponding to the next subset can be obtained using exactly two ℓ-bit vector additions. As a result, the total number of ℓ-bit vector additions required for all the subsets becomes $2 \cdot L_1$. This is an improvement over the naive method if $p > 5$. Similar efficiency improvements are obtained for the computations of $\mathbf{b}_1$, $\mathbf{a}_2$ and $\mathbf{b}_2$. Even though the use of Chase's sequence is advantageous only for $p > 5$, for the sake of obtaining a single expression for the time complexity, we will assume $2 \cdot L_1$ vector additions are required even for $p < 5$, and similarly for the computations of $\mathbf{b}_1$, $\mathbf{a}_2$ and $\mathbf{b}_2$. This makes the time complexity estimates slightly worse for $2 \leq p \leq 5$.

Early abort: The final check for a solution is to compare the weight of $\mathbf{x} = \mathbf{b}_1 + \mathbf{b}_2 + \mathbf{v}$ with $\omega - p$. Note that the length of $\mathbf{x}$ is $r - \ell$, which in general is substantially greater than $\omega - p$. This observation forms the basis of the technique of early abort in [20]. Instead of first computing $\mathbf{x}$ and then comparing its weight to $\omega - p$, it is faster to compute $\mathbf{x}$ incrementally and abort once the weight of the partially computed $\mathbf{x}$ exceeds $\omega - p$. If $\mathbf{x}$ does not correspond to a solution, it is reasonable to assume that it will behave like a random binary string of length $r - \ell$. So the first $2(\omega - p + 1)$ positions is likely to have weight $\omega - p + 1$ and such a vector $\mathbf{x}$ can be discarded without computing the other bits. For most vectors $\mathbf{x}$, this brings down the cost of checking $\mathsf{wt}(\mathbf{x}) = \omega - p$ from $r - \ell$ bit operations to an expected number $2(\omega - p + 1)$ of bit operations.

There are several other sophisticated techniques to improve both the linear algebra step and the search step of Stern's algorithm [20, 18]. All of these techniques also apply to the generalised Stern's algorithm. For the sake of simplicity we do not include the effect of these techniques in the present analysis. Nonetheless, time estimates for the variants of Classic McEliece obtained from our simple model are quite close to the time estimates obtained using the more detailed techniques of [18]; see Section 5.2.

Time complexity. We estimate the number of bit operations required by Algorithm 4.

The quantity L_1 is defined in Algorithm Srch. Recall that $c = k + \lambda$ and let

$$L_2 \;=\; \binom{\lceil c/2 \rceil}{\lceil p/2 \rceil}. \tag{4.5}$$

Since the algorithm is probabilistic, first we calculate the success probability of the algorithm in a single iteration. We assume that there is one solution $\mathbf{e}_0$ (of weight ω) to the ISD instance $(\mathbf{H}_0, \mathbf{s}_0, \omega)$. By success probability we mean the probability of the event that the algorithm returns this solution. (If there are more than one solutions, then the success probability will be higher.)

We assume that the permutation matrix $\mathbf{P}$ is chosen uniformly and independently in each iteration. So in each iteration, a uniform random permutation is applied to the columns of the parity check matrix $\mathbf{H}_0$. The total number of such permutations is $n!$. Let $\mathcal{P}$ be the set of 'good' permutations, i.e. if any permutation from $\mathcal{P}$ is applied to the columns of $\mathbf{H}_0$, then a solution is obtained in the search step. So the success probability π of the search step of a single iteration is

$$\pi \;=\; \frac{\#\mathcal{P}}{n!}. \tag{4.6}$$

The set $\mathcal{P}$ can be constructed in the following manner. Let $i_1, \ldots, i_w$ be the one-positions of $\mathbf{e}_0$ (i.e. the positions where $\mathbf{e}_0$ is 1). Call the other positions of $\mathbf{e}_0$ to be zero-positions. It is helpful to visualise the construction of the permutations in $\mathcal{P}$ as distributing the one-positions and the zero-positions to the cells of an array of length n. Distribute the first $\lfloor p/2 \rfloor$ one-positions to a subset of the cells $1, \ldots, \lfloor c/2 \rfloor$ of size $\lfloor p/2 \rfloor$ in a manner such that these cells form some subset S in $\mathcal{L}$ (there are $\#\mathcal{L}$ ways to make this distribution); distribute the next $\lceil p/2 \rceil$ one-positions to some subset of the cells $(\lfloor c/2 \rfloor + 1), \ldots, c$ of size $\lceil p/2 \rceil$ (there are L_2 ways to make this distribution); distribute the remaining $\omega - p$ one-positions to some subset of the cells $(c+1), \ldots, n$ of size $\omega - p$ (there are $\binom{n-k-\ell}{\omega-p}$ ways to make this distribution); and then fill the remaining $n - \omega$ cells with the zero-positions in some particular order. This fixes the positions for the one-positions and the zero-positions in the array. This fixing can be done in

$$\#\mathcal{L} \cdot L_2 \cdot \binom{n-k-\ell}{\omega-p}$$

ways. Now permute the cells filled with the one-positions among themselves and also permute the cells filled with the zero-positions among themselves, which can be done in $\omega!(n-\omega)!$ ways. So the size of $\mathcal{P}$ is

$$\#\mathcal{P} \;=\; \#\mathcal{L} \cdot L_2 \cdot \binom{n-k-\ell}{\omega-p} \cdot \omega!(n-\omega)!. \tag{4.7}$$

Algorithm Srch ensures that the size of $\mathcal{L}$ is $\lceil L_1^{\delta} \rceil$. Using (4.6) and (4.7), we have

$$\pi \;=\; \frac{\lceil L_1^{\delta} \rceil \cdot L_2 \cdot \binom{n-k-\ell}{\omega-p}}{\binom{n}{\omega}}. \tag{4.8}$$

Let N be the number of iterations required to obtain success. Then N follows a geometric distribution with parameter π. Consequently,

$$\mathbb{E}[N] \;=\; \frac{1}{\pi}. \tag{4.9}$$

Let X_i be the random variable whose value is given by the number of bit operations performed in the i-th iteration. The total number of bit operations is $X_1 + X_2 + \cdots + X_N$. Let T be the expected value of the total number of bit operations. Under the heuristic assumption that the X_i's are independent and identically distributed, and N is independent of the X_i's, by Wald's equation (see Page 300 of [84]),

$$T \;=\; \mathbb{E}[X_1 + X_2 + \cdots + X_N] = \mathbb{E}[X_1] \cdot \mathbb{E}[N]. \tag{4.10}$$

Next we obtain an estimate for $\mathbb{E}[X_1]$, i.e. the expected number of bit operations in each iteration. This has two components, the number of bit operations due to linear algebra step and the number of bit operations due to the search step. Denoting by T_{LA} and T_{SR} the expected number of bit operations required for linear algebra and search steps respectively, we have

$$\mathbb{E}[X_1] \;=\; T_{\mathrm{LA}} + T_{\mathrm{SR}}. \tag{4.11}$$

Using (4.9), (4.10) and (4.11), we obtain

$$T \;=\; \frac{1}{\pi} \cdot (T_{\mathrm{LA}} + T_{\mathrm{SR}}) = \frac{\binom{n}{\omega}}{\lceil L_1^{\delta} \rceil \cdot L_2 \cdot \binom{n-k-\ell}{\omega-p}} \cdot (T_{\mathrm{LA}} + T_{\mathrm{SR}})\,. \tag{4.12}$$

Remark 4. *The above analysis ignores the effect of E_1, i.e. the event that the linear algebra step succeeds (see Remark 2). We briefly consider this effect. Let E_2 be the event that the search step succeeds and so $\pi = \Pr[E_2]$. Let M be a random variable whose value is the number of iterations required by Algorithm 4 to achieve success. For $i = 1, \ldots, M$, let T_i be the binary valued random variable which takes the value 1 if and only if E_1 occurs in the i-th step. So $\mathbb{E}[T_i] = \pi_1$. Let $N = \sum_{i=1}^{M} T_i$, i.e. N is the number of times the search step is repeated until success is obtained (i.e. E_2 occurs), and so $\mathbb{E}[N] = 1/\pi$. Hence, by an application of Wald's equation, $\mathbb{E}[M] = (\pi \cdot \pi_1)^{-1}$. Let Y_i and Z_i be random variables whose values are the numbers of bit operations required for the linear algebra and the search step respectively. As above, let X_i be the number of bit operations required in the i-th step, and so $X_i = Y_i + T_i Z_i$. Note that $\mathbb{E}[Y_i] = T_{\mathrm{LA}}$ and $\mathbb{E}[Z_i] = T_{\mathrm{SR}}$. Letting T denote the expected value of the total number of bit operations, we have $T = \mathbb{E}[\sum_{i=1}^{M} X_i] = \mathbb{E}[\sum_{i=1}^{M} Y_i] + \mathbb{E}[\sum_{i=1}^{M} T_i Z_i]$. Applying Wald's equation separately to the two terms and heuristically assuming that T_i and Z_i are independent, we obtain $T = \pi^{-1} \cdot (T_{\mathrm{LA}} \cdot \pi_1^{-1} + T_{\mathrm{SR}})$. Note the difference with the expression for T given by (4.12). This difference, however, does not cause noticeable difference in the concrete time estimates and so for simplicity, we consider T to be given by (4.12).*

Next we describe estimates of T_{LA} and T_{SR}. If $\lambda \leq 0$, then the last ℓ columns of the matrices $\mathbf{C}$ and $\mathbf{D}$ in (4.2) are $\mathbf{I}_\ell$ and the all-zero matrices respectively. This corresponds

to the row operations required in Stern's algorithm. For $0 < \lambda \leq \ell$, the matrices $\mathbf{C}$ and $\mathbf{D}$ are smaller and the linear algebra step possibly requires less number of bit operations. For simplicity of analysis, we assume that the number of bit operations required for the linear algebra step is equal to that of Stern's algorithm. Following [20], we estimate the expected number of bit operations required for the linear algebra step in Stern's algorithm to be

$$T_{\text{LA}} \;=\; \frac{k^2(n-k)(n-k-1)(3n-k)}{4n^2}. \tag{4.13}$$

In the search step, bit operations are required to compute the quantities $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{b}_1$, $\mathbf{b}_2$ and $\mathbf{b}_1+\mathbf{b}_2+\mathbf{v}$. Using the early abort technique (see Section 4.2) requires computing only the first $2(\omega-p+1)$ bits of the last three quantities. Using Chase's sequence (again see Section 4.2) the computation of all the $\lceil L_1^\delta \rceil$ $\mathbf{a}_1$'s and $\mathbf{b}_1$'s require a total of $(2\ell + 4(\omega-p+1))\lceil L_1^\delta \rceil$ bit operations. Similarly, using Chase's sequence the computation of all the L_2 $\mathbf{a}_2$'s and $\mathbf{b}_2$'s require a total of $(2\ell+4(\omega-p+1))L_2$ bit operations. Finally, we consider the number of bit operations in the collision phase. We make the heuristic assumption that the $\mathbf{a}_1$'s that arise due to the different choices of S are independent and uniformly distributed ℓ-bit strings. So for any fixed ℓ-bit string $\mathbf{x}$, the probability it arises as $\mathbf{a}_1$ due to any particular choice of S is equal to $1/2^\ell$; consequently, the total number of times $\mathbf{x}$ occurs as a first component in the list $\mathcal{L}$ follows the binomial distribution with parameters $1/2^\ell$ and $\#\mathcal{L}$. So the expected number of times $\mathbf{x}$ occurs as a first component in the list $\mathcal{L}$ is $\#\mathcal{L}/2^\ell = \lceil L_1^\delta \rceil / 2^\ell$. It then follows that for each of the L_2 $\mathbf{a}_2$'s (which are also ℓ-bit strings) generated in the collision phase, on an average there will be about $\lceil L_1^\delta \rceil / 2^\ell$ $\mathbf{a}_1$'s in $\mathcal{L}$ such that the condition $\mathbf{a}_1+\mathbf{u} = \mathbf{a}_2$ holds. So the computation of $\mathbf{b}_1 + \mathbf{b}_2 + \mathbf{v}$ has to be done a total of about $L_2 \cdot (\lceil L_1^\delta \rceil / 2^\ell)$ times. Using the early abort technique, this requires about $L_2 \cdot (\lceil L_1^\delta \rceil / 2^\ell) \cdot (2(\omega-p+1))$ bit operations. Putting the calculations together, we have

$$\begin{aligned} T_{\text{SR}} \;\approx\; & (2\ell + 4(\omega-p+1))\lceil L_1^\delta \rceil \\ & +(2\ell + 4(\omega-p+1))L_2 + L_2 \cdot (\lceil L_1^\delta \rceil / 2^\ell) \cdot (2(\omega-p+1)). \end{aligned} \tag{4.14}$$

Memory complexity. An instance of the computational syndrome decoding problem is a triplet $(\mathbf{H}_0, \mathbf{s}_0, \omega)$. So any ISD algorithm has to store the matrix $\mathbf{H}_0$ and the syndrome $\mathbf{s}_0$. Let M_{mat} be the number of bits required to store $\mathbf{H}_0$ and $\mathbf{s}_0$. Then

$$M_{\text{mat}} \;=\; (n-k)(n+1). \tag{4.15}$$

The additional memory requirement of Algorithm 4 arises from the memory requirement of Algorithm Srch, which needs to store the list $\mathcal{L}$. The size of $\mathcal{L}$ is $\lceil L_1^\delta \rceil$. Each entry of $\mathcal{L}$ is of the form $(\mathbf{a}_1, \mathbf{b}_1, S)$, where $\mathbf{a}_1$ is an ℓ-bit vector, $\mathbf{b}_1$ is an $(r-\ell)$-bit vector and M is a subset of $[\lfloor (k+\lambda)/2 \rfloor]$ of size $\lfloor p/2 \rfloor$. Since we consider the early abort technique, only the first $2(\omega-p+1)$ bits of $\mathbf{b}_1$ are stored. Let M_{lst} be the number of bits required to store $\mathcal{L}$. Then

$$M_{\text{lst}} \;=\; \lceil L_1^\delta \rceil \cdot (\ell + 2(\omega-p+1) + \lfloor p/2 \rfloor \cdot \lceil \log_2 \lfloor (k+\lambda)/2 \rfloor \rceil). \tag{4.16}$$

The total memory required by Algorithm 4 is M, where

$$M \;=\; M_{\text{mat}} + M_{\text{lst}}. \tag{4.17}$$

Cost of memory access. If M is large, then accessing memory will require non-negligible time. The logarithmic memory access cost model has been suggested in previous works [53]. In this model, the time estimate is increased by a factor which is equal to the logarithm to the base two of the memory estimate. So the time estimate T_{ma} taking logarithmic memory access cost into consideration is given by

$$T_{\text{ma}} = T \cdot \log_2 M. \tag{4.18}$$

Remark 5. *Estimating the cost of memory access using* (4.18) *is rather adhoc. For one thing it assumes that each bit operation requires a memory access which is not the case. Secondly, a large fraction of the bit operations are on the matrix* $\mathbf{H}$ *and not on the list* $\mathcal{L}$*. Compared to* $\mathbf{H}$*, the list* $\mathcal{L}$ *can require much more memory to store. So assigning the same cost of memory access to memory operations on* $\mathbf{H}$ *and* $\mathcal{L}$ *may not be justified.*

Remark 6. *A Boolean circuit model based analysis of time estimates of* ISD *algorithms has been performed in [18]. Such an analysis inherently incorporates cost of memory access which would otherwise be ignored in an analysis assuming constant time for memory access. Note, however, that incorporating logarithmic memory access cost as in* (4.18) *captures memory access cost in a manner which is different from that in [18].*

Complexity of Prange's, Lee-Brickell's and Leon's algorithms. Let C_1, C_2 and C_3 be the expected number of bit operations required by Prange's, Lee and Brickell's and Leon's algorithms respectively. We have

$$C_1 = \frac{\binom{n}{\omega}}{\binom{n-k}{\omega}} \cdot (T_{\text{LA}} + 2(\omega+1)), \tag{4.19}$$

$$C_2 = \frac{\binom{n}{\omega}}{\binom{k}{p}\binom{n-k}{\omega-p}} \cdot \left(T_{\text{LA}} + \binom{k}{p} \cdot (4(\omega - p + 1)) \right), \tag{4.20}$$

$$C_3 = \frac{\binom{n}{\omega}}{\binom{k}{p}\binom{n-k-\ell}{\omega-p}} \cdot \left(T_{\text{LA}} + \binom{k}{p} \cdot \left(2\ell + \frac{2p(\omega-p+1)}{2^\ell} \right) \right). \tag{4.21}$$

All the three algorithms store only $\mathbf{H}_0$ and $\mathbf{s}_0$ and hence the number of bits required to be stored is M_{mat}.

4.3 Application to Classic McEliece

The expected security categories of the five variants are given in Section 7 of the Classic McEliece specification [19]. For the five variants, our abbreviations of the names, the values of n, k and ω for the five variants, and their categories are shown in Table 4.1. Section 2.2.3 of the Classic McEliece specification [19], states that the public key is an $r \times k$ binary matrix. Table 4.1 shows $\log_2 P$, where $P = r \cdot k$ is the size of the public key in bits. An ISD instance is given by $(\mathbf{H}_0, \mathbf{s}_0, \omega)$. So M_{mat} bits are required to store $\mathbf{H}_0$ and $\mathbf{s}_0$. The values of $\log_2 M_{\text{mat}}$ for the variants of the Classic McEliece are shown in Table 4.1.

Table 4.1: Parameters for Classic McEliece and the corresponding values of $\log_2 P$ and $\log_2 M_{\text{mat}}$.

category	name	n	k	ω	$\log_2 P$	$\log_2 M_{\text{mat}}$
1	`mceliece-3488-064` (`m3488`)	3488	2720	64	20.99	21.35
3	`mceliece-4608-096` (`m4608`)	4608	3360	96	22.00	22.46
5	`mceliece-6688-128` (`m6688`)	6688	5024	128	22.00	23.41
	`mceliece-6960-119` (`m6960`)	6960	5413	119	22.00	23.36
	`mceliece-8192-128` (`m8192`)	8192	6528	128	23.37	23.70

For Algorithm 4, instead of using the maximum possible ranges of values of parameters given by (4.1), we have restricted to the following ranges of values of the parameters. The concrete results obtained using these choices indicate that there are no interesting TMTO points for values of parameters outside these ranges.

$$0 \leq \ell \leq 100, \qquad 2 \leq p \leq 30, \qquad -\ell \leq \lambda \leq \ell, \qquad \delta = 1 - 0.025 \cdot i,\ i = 0, \ldots, 12. \tag{4.22}$$

The optimal time and memory complexity estimates of Algorithm 4 are same as the optimal time and memory complexity estimates of Dumer's algorithm for the same ranges of parameters p and ℓ. In the next chapter (Chapter 5) we shall introduce a new concept called *Effective Set of Time/Memory Trade-off Points* which shows us that Algorithm 4 gives us more number of trade-off points than Stern's and Dumer's algorithms when the five variants of Classic McEliece are considered.

4.4 Summary

We have proposed a generalised version of Stern's ISD algorithm using a new parameter set. By appropriately selecting the values of the parameters of the new algorithm, it is possible to obtain Stern's, Dumer's, and Bernstein and Chou's algorithms as special cases. We have also discussed the time complexity and memory complexity of the new algorithm.

Chapter 5

Computing Effective Time/Memory Set

In section 4.2 (please see equation 4.1) of Chapter 4 the ranges of the various parameters have been mentioned. Varying the parameters of an ISD algorithm provides a large number of TMTO points. It is quite difficult to directly analyse the entire set of all TMTO points. To tackle this issue we introduce the notion of a set of effective TMTO points of an ISD algorithm with respect to a range of values of the parameters of the algorithm. Such a set succinctly and uniquely captures the entire TMTO landscape at only a minor loss in precision. Further, we describe a method to compute a set of effective TMTO points for any ISD algorithm.

As an application, we have obtained sets of effective TMTO points corresponding to the five variants of the Classic McEliece cryptosystem for the generalisation of Stern's algorithm that we propose as well as for Stern's, Dumer's and Bernstein and Chou's algorithms. The results show that Dumer's and Bernstein and Chou's algorithms do not provide any interesting TMTO points beyond what is achieved by Stern's algorithm.

On the other hand, the sets of effective TMTO points obtained from the generalisation that we introduce are about twice the sizes of the corresponding sets of effective TMTO points obtained from Stern's algorithm. Further, in each case, the set of effective TMTO points obtained from the new generalised algorithm essentially subsumes the corresponding set obtained from Stern's algorithm. In particular, there are certain TMTO points which are achieved by the new generalised algorithm, but not by either of Stern's, Dumer's and Bernstein and Chou's algorithms. The TMTO points themselves are quite interesting. For example, these points show that in certain cases, by letting the time estimates increase by factors which are at most 2, it is possible to reduce the memory requirements by factors of about 2^8 to 2^{10}.

By obtaining sets of effective TMTO points, we are able to address the question of what time complexity can be achieved without requiring memory much larger than the public key size. For `mceliece-4608-096`, the size of the public key is about 2^{22} and the NIST target is 2^{207} gates. For this variant, it is possible to obtain TMTO points having time complexities less than 2^{207} with memory requirement only about 1.5 times the size of the public key for both the cases of constant memory access cost and logarithmic memory access cost. For

`mceliece-6688-128` and `mceliece-6960-119`, the sizes of the public key are also about 2^{22} bits and the NIST target for both is 2^{272} gates. With constant memory access cost, time complexities of about 2^{265} can be obtained for both these variants. The corresponding memory complexities are $2^{57.97}$ and $2^{67.37}$ respectively which are much larger than the size of the public key. By choosing values of the parameters appropriately, we find that it is possible to obtain time complexities slightly less than 2^{272} while keeping the memory complexities to be less than 2^{30}. While the memory complexities are still larger than the size 2^{22} of the public key, they are not too large. If, on the other hand, logarithmic cost of memory access is incorporated into the time estimates, then with memory restricted to be less than 2^{30} bits, the minimum time estimates for both the variants turn out to be more than the target value of 2^{272} by factors which are less than 8. For the other two variants of the Classic McEliece cryptosystem, we do not obtain any TMTO point, even with constant memory access cost, whose time complexity is less than what is required for NIST classification.

This chapter is based on the second contribution of the work in [23].

5.1 A set of effective TMTO points

We describe what we mean by a set of effective TMTO points and a procedure to compute such a set. Our initial description is based on Algorithm 4. Later we mention how the procedure can be applied to any ISD algorithm.

For fixed values of n, k and ω, it is possible to evaluate the expressions for T and M given by (4.12) and (4.17) respectively for every possible choice of the values of the parameters ℓ, p, λ and δ. This leads to a large number of TMTO points (T, M). For example, using the range of parameters given by (5.3) in Section 5.2 leads to more than 3 million TMTO points. There are, however, large clusters of values of T and M. As an example, for the variant `mceliece-3488-064` of Classic McEliece, we have $n = 3488$, $k = 2720$ and $\omega = 64$. In this case, the minimum value of $\log_2 T$ required by Algorithm 4 is 147.70988 and there are 819 values which are less than 148, the largest of which is 147.99983. Similar clustering also occurs for $\log_2 M$. Based on these observations, we define two time complexities T and T' to be equivalent if $\lceil \log_2 T \rceil = \lceil \log_2 T' \rceil$, and two memory complexities M and M' to be equivalent if $\lceil \log_2 M \rceil = \lceil \log_2 M' \rceil$. Extending to TMTO points, we define two TMTO points (T, M) and (T', M') to be equivalent if $\lceil \log_2 T \rceil = \lceil \log_2 T' \rceil$ and $\lceil \log_2 M \rceil = \lceil \log_2 M' \rceil$. So the time and memory complexities of a TMTO point differ from the respective complexities of an equivalent TMTO point by factors which are less than 2.

For fixed values of n, k and ω, let $\mathcal{T}_0$ be the set of all tuples

$$(\lceil \log_2 T \rceil, \lceil \log_2 M \rceil, \ell, p, \lambda, \delta, \log_2 T, \log_2 M) \tag{5.1}$$

corresponding to a pre-determined range of values of the parameters ℓ, p, λ and δ. The list $\mathcal{T}_0$ captures all the TMTO points arising from the chosen range of values of the parameters. We say that a TMTO point (T, M) is represented in $\mathcal{T}_0$ if $(\log_2 T, \log_2 M)$ occurs as the last two components of some tuple in $\mathcal{T}_0$. Let $\mathcal{V}$ be a non-empty set of TMTO points represented in $\mathcal{T}_0$ satisfying the following two properties.

1. *Minimality:* If $(T, M) \neq (T', M')$ are in $\mathcal{V}$, then $\lceil \log_2 T \rceil \neq \lceil \log_2 T' \rceil$, $\lceil \log_2 M \rceil \neq \lceil \log_2 M' \rceil$, and either $\lceil \log_2 T \rceil > \lceil \log_2 T' \rceil$ or $\lceil \log_2 M \rceil > \lceil \log_2 M' \rceil$.
2. *Completeness:* If (T', M') is represented in $\mathcal{T}_0$, then there is a point (T, M) in $\mathcal{V}$ such that $\lceil \log_2 T \rceil \leq \lceil \log_2 T' \rceil$ and $\lceil \log_2 M \rceil \leq \lceil \log_2 M' \rceil$.

We say that the set $\mathcal{V}$ is a *set of effective TMTO points* with respect to the chosen range of parameters. A consequence of the first condition is that two distinct points in $\mathcal{V}$ are inequivalent. Let

$$\mathsf{CL}(\mathcal{V}) \quad = \quad \{(\lceil \log_2 T \rceil, \lceil \log_2 M \rceil) : (T, M) \in \mathcal{V}\}. \tag{5.2}$$

Proposition 1. *If $\mathcal{V}$ and $\mathcal{V}'$ are two sets of effective TMTO points for the same ranges of values of the parameters, then $\mathsf{CL}(\mathcal{V}) = \mathsf{CL}(\mathcal{V}')$.*

Proof. Suppose $(\lceil \log_2 T' \rceil, \lceil \log_2 M' \rceil)$ is in $\mathsf{CL}(\mathcal{V}')$ corresponding to some point (T', M') in $\mathcal{V}'$. Since $\mathcal{V}'$ is a set of effective TMTO points, by definition, (T', M') is represented in $\mathcal{T}_0$. Since $\mathcal{V}$ is also a set of effective TMTO points, by completeness of $\mathcal{V}$, there is a point (T, M) in $\mathcal{V}$ which is represented in $\mathcal{T}_0$ satisfying $\lceil \log_2 T \rceil \leq \lceil \log_2 T' \rceil$ and $\lceil \log_2 M \rceil \leq \lceil \log_2 M' \rceil$. Further, $(\lceil \log_2 T \rceil, \lceil \log_2 M \rceil)$ is in $\mathsf{CL}(\mathcal{V})$. Since (T, M) is represented in $\mathcal{T}_0$ and $\mathcal{V}'$ is a set of effective TMTO points, by completeness of $\mathcal{V}'$, there is a point (T'', M'') in $\mathsf{CL}(\mathcal{V}')$ satisfying $\lceil \log_2 T'' \rceil \leq \lceil \log_2 T \rceil$ and $\lceil \log_2 M'' \rceil \leq \lceil \log_2 M \rceil$. So it follows that $\lceil \log_2 T'' \rceil \leq \lceil \log_2 T' \rceil$ and $\lceil \log_2 M'' \rceil \leq \lceil \log_2 M' \rceil$. By minimality of $\mathcal{V}'$, the condition in the previous sentence is only possible if $\lceil \log_2 T'' \rceil = \lceil \log_2 T' \rceil$ and $\lceil \log_2 M'' \rceil = \lceil \log_2 M' \rceil$, which implies that $\lceil \log_2 T \rceil = \lceil \log_2 T' \rceil$ and $\lceil \log_2 M \rceil = \lceil \log_2 M' \rceil$, and so $(\lceil \log_2 T' \rceil, \lceil \log_2 M' \rceil)$ is in $\mathsf{CL}(\mathcal{V})$. This shows that $\mathsf{CL}(\mathcal{V}')$ is a subset of $\mathsf{CL}(\mathcal{V})$. Reversing the argument, we have $\mathsf{CL}(\mathcal{V})$ to be a subset of $\mathsf{CL}(\mathcal{V}')$. □

Proposition 1 shows that a set of effective TMTO points uniquely captures the entire landscape of all TMTO points up to loss of precision by factors which are less than 2. Later we will see examples which show that a set of effective TMTO points can be much smaller than the size of all TMTO points.

We describe a method to compute a set of effective TMTO points. The idea is to progressively process a list of tuples $\mathcal{T}$ which is initially set to be equal to $\mathcal{T}_0$. The following steps are then performed successively on $\mathcal{T}$.

1. *Sorting:* Perform an ascending order sort of the tuples in $\mathcal{T}$, where the usual lexicographic ordering of tuples is assumed, i.e. a tuple is considered to be less than another if for some $i \geq 1$, the first $i-1$ components of the two tuples are equal and the i-th component of the first tuple is less than that of the second.
2. *First filtering:* Perform a filtering on $\mathcal{T}$ to ensure that for any particular value of $\lceil \log_2 T \rceil$, only the first tuple with the given value is retained while all other tuples with the same value of $\lceil \log_2 T \rceil$ are dropped.
3. *Second filtering:* Perform a filtering on $\mathcal{T}$ to ensure that for any particular value of $\lceil \log_2 M \rceil$, only the first tuple with the given value is retained while all other tuples with the same value of $\lceil \log_2 M \rceil$ are dropped.
4. *Pruning:* Discard all tuples from $\mathcal{T}$ starting from the point where $\lceil \log_2 T \rceil$ increases but $\lceil \log_2 M \rceil$ does not decrease.

Let $\mathcal{T}_1$ be the final state of $\mathcal{T}$ and let $\mathcal{U}$ be the set of all TMTO points (T, M) which are represented in $\mathcal{T}_1$. We have the following result.

Proposition 2. *$\mathcal{U}$ is a set of effective TMTO points.*

Proof. Since $\mathcal{T}_1$ is obtained from $\mathcal{T}_0$ by removing tuples, any tuple in $\mathcal{T}_1$ is also in $\mathcal{T}_0$. So any TMTO point in $\mathcal{U}$ is represented in $\mathcal{T}_0$.

Suppose (T, M) and (T', M') are two TMTO points in $\mathcal{U}$. The first and the second filtering steps ensure that $\lceil \log_2 T \rceil \neq \lceil \log_2 T' \rceil$ and $\lceil \log_2 M \rceil \neq \lceil \log_2 M' \rceil$ respectively. The pruning step ensures that if $\lceil \log_2 T \rceil < \lceil \log_2 T' \rceil$, then $\lceil \log_2 M \rceil > \lceil \log_2 M' \rceil$, as otherwise the tuple representing (T', M') would be dropped in the pruning step. This shows the minimality of $\mathcal{U}$.

Suppose (T', M') is a TMTO point represented in $\mathcal{T}_0$. If it is not in $\mathcal{U}$, then the tuple which represents it in $\mathcal{T}_0$ was dropped in one of the two filtering steps or in the pruning step. If it was dropped in the first filtering step, then there is a TMTO point (T, M) in $\mathcal{U}$ such that $\lceil \log_2 T \rceil = \lceil \log_2 T' \rceil$ and $\lceil \log_2 M \rceil \leq \lceil \log_2 M' \rceil$. If it was dropped in the second filtering step, then there is a TMTO point (T, M) in $\mathcal{U}$ such that $\lceil \log_2 T \rceil \leq \lceil \log_2 T' \rceil$ and $\lceil \log_2 M \rceil = \lceil \log_2 M' \rceil$. Finally, if it was dropped in the pruning step, then there is a TMTO point (T, M) in $\mathcal{U}$ such that $\lceil \log_2 T \rceil < \lceil \log_2 T' \rceil$ and $\lceil \log_2 M \rceil < \lceil \log_2 M' \rceil$. This shows the completeness of $\mathcal{U}$. □

Setting $\delta = 1$ in the code provides a set of effective TMTO points for the Bernstein-Chou generalisation of Stern's algorithm; setting $\lambda = 0$ and $\delta = 1$ in the code provides a set of effective TMTO points for Stern's algorithm; setting $\lambda = \ell$ and $\delta = 1$ in the code provides a set of effective TMTO points for Dumer's algorithm.

The discussion above for obtaining a set of effective TMTO points was with reference to Algorithm 4. Changing the procedure to any other ISD algorithm is simply to change the parameters and to use the appropriate expressions (or algorithms) to compute the time and memory complexities of the algorithm. This leads to changing (5.1) and affects the construction of the initial state $\mathcal{T}_0$ of $\mathcal{T}$. The sorting, filtering and pruning steps are applied to $\mathcal{T}$ as described above. A set $\mathcal{U}$ of effective TMTO points (with respect to the chosen range of values of the parameters) can then be defined from the final state $\mathcal{T}_1$ of $\mathcal{T}$ in the same manner as described above.

Comparison to previous TMTO analysis of ISD *algorithms.* The TMTO analysis considered in [53] is of the following type. Fix an upper bound M on the memory complexity and then obtain the minimum time required by the algorithm utilising at most M bits of memory. Let us call this a memory bounded approach to TMTO analysis. Such an approach provides less information in comparison to the analysis using the notion of an effective set of TMTO points. We illustrate this using an example. Consider Table 5.2a which provides certain TMTO points for `mceliece-3488-064` achieved by Algorithm 4 using the notion of effective set of TMTO points. Now consider the memory bounded analysis. Suppose we fix the memory bound M to be 2^{50} and ask what is the minimum time that can be achieved by Algorithm 4 utilising at most 2^{50} bits of memory? From Table 5.2a the answer is $2^{147.93}$. However, from the table we also find that the time $2^{147.93}$ can be achieved using $2^{44.55}$ bits of memory. Similarly, if we fix M to be 2^{40}, then the memory bounded analysis will provide

minimum time estimate of $2^{148.93}$, while the analysis based on effective set of TMTO points will provide the additional information that the time estimate of $2^{148.93}$ can be achieved using $2^{34.63}$ bits of memory. In both the above cases, the actual memory requirements $2^{44.55}$ or $2^{34.63}$ are lower than the upper bounds 2^{50} and 2^{40} respectively. So while the memory bounded analysis obtains the minimum time subject to an upper bound on the memory, it does not yield the actual memory that is required to achieve the minimum time. In contrast, the analysis based on the notion of effective set of TMTO points provides for each time estimate the minimum memory required to achieve the time estimate. More generally, due to the completeness property an effective set of TMTO points captures (up to a minor loss in precision) the entire landscape of TMTO points. In particular, for any TMTO point (T', M') obtained using a memory bounded analysis, the completeness property assures us that there is a TMTO point (T, M) in an effective set of TMTO points such that $\lceil \log_2 T \rceil \leq \lceil \log_2 T' \rceil$ and $\lceil \log_2 M \rceil \leq \lceil \log_2 M' \rceil$.

Justifications for the estimates in (4.19)**,** (4.20) **and** (4.21)**.** A generalisation of Algorithm 4 has been outlined in Section 4.2 which provides Prange's, Lee-Brickell's and Leon's algorithms as special cases. The analysis of time and memory complexities of Algorithm 4 can also be easily generalised to obtain expressions for the time and memory complexities of the unified algorithm sketched in Section 4.2. The expressions for C_1, C_2 and C_3 can then be obtained as special cases. Instead of adopting this approach, we briefly sketch the arguments required to directly obtain the required expressions.

As in the case of Algorithm 4, applying (4.10), the expected number of bit operations is given by the product of the expected number of bit operations in each iteration and the inverse of the success probability. The success probabilities of the three algorithms can be obtained using standard arguments and we refer to [9] for the details. Each iteration consists of the linear algebra step and the search step. The number of bit operations in the linear algebra step is the same as that in Stern's algorithm which we take to be T_{LA}. It only remains to consider the expected number of bit operations in the search step. The difference with the analysis in [9] arises from our use of Chase's sequence and the early abort technique.

In Prange's algorithm, the search step is simply to check that the weight of $\mathbf{s}$ is ω. Using the early abort technique, this requires about $2(\omega + 1)$ bit operations in almost all cases.

In the Lee-Brickell algorithm, the search step computes all possible p-column combinations of $\mathbf{A}$ (which in this case is an $r \times k$ matrix) and checks whether the sum of $\mathbf{s}$ and any such p-column combination $\mathbf{x}$ is equal to $\omega - p$. Using Chase's sequence, the next p-column combination from the previous one can be generated using two vector additions. Further, using the early abort technique, it is sufficient to compute only the first $2(\omega - p + 1)$ bits $\mathbf{s} + \mathbf{x}$ in almost all cases. So the expected number of bit operations is $4(\omega - p + 1)$.

In Leon's algorithm, the search step computes all possible p-column combinations of $\mathbf{A}$ (which in this case is an $\ell \times k$ matrix) and checks whether any such $\mathbf{x}$ is equal to $\mathbf{u}$. Using Chase's sequence, the number of bit operations required to generate the next p-column combination from the previous one is 2ℓ. If some $\mathbf{x}$ is equal to $\mathbf{u}$, then the corresponding p-column combination $\mathbf{y}$ of $\mathbf{B}$ is computed and it is checked whether the weight of $\mathbf{y} + \mathbf{v}$ is equal to $\omega - p$. We make the usual heuristic assumption that the $\mathbf{x}$'s generated by all the p-column combinations of $\mathbf{A}$ are independent and uniformly distributed. Under this assumption, any particular $\mathbf{x}$ is equal to $\mathbf{u}$ with probability $1/2^{\ell}$ and so the number of $\mathbf{x}$'s

Table 5.1: Concrete time estimates of Leon's algorithm. The entries in Column-A are estimates of either Leon's or Lee-Brickell's algorithms.

	$\log_2 C_3$	$\log_2(C_3 \cdot \log_2 M_{\text{mat}})$	A [18]
`m3488`	157.91	162.33	159.93
`m4608`	200.71	205.20	202.30
`m6688`	278.28	282.82	279.88
`m6960`	279.33	283.87	-
`m8192`	316.05	320.62	317.66

which are equal to $\mathbf{u}$ follows the binomial distribution with parameters $1/2^\ell$ and $\binom{k}{p}$. So the expected number of times $\mathbf{x}$ is equal to $\mathbf{u}$ is equal to $\binom{k}{p}/2^\ell$. For each such match, the sum $\mathbf{y}+\mathbf{v}$ is to be computed. Using the early abort technique, it is sufficient to compute the first $2(\omega - p + 1)$ bits of this sum in almost all cases. This gives us the required expression.

Concrete estimates of time complexity of Leon's algorithm for Classic McEliece. Table 5.1 provides the minimum value of C_3, the expected number of bit operations required by Leon's algorithm, where the minimum is taken over the appropriate ranges of ℓ and p. The column headed by $\log_2(C_3 \cdot \log_2 M_{\text{mat}})$ reports the time estimates of Leon's algorithm adjusted for logarithmic memory access cost. We used a Python script to compute the minimum values of C_3 and $\log_2(C_3 \cdot \log_2 M_{\text{mat}})$. The entries in the column headed by A are the binary logarithms of the minimum time estimates of Lee-Brickell or Leon's algorithm from [18]; the work does not specify the particular algorithm corresponding to a particular time estimate. While [18] captures a wide range of optimisations as well as costs, including memory access costs, it may be noted that the values in column A are in-between our estimates of Leon's algorithm with constant and logarithmic memory access costs, which are quite close.

5.2 Application to Classic McEliece

The NIST call for proposals [87] for post-quantum cryptosystems outlines five categories. Of these, Categories 1, 3 and 5 require cryptosystems to be secure under attacks using 2^{143}, 2^{207} and 2^{272} classical gates respectively.

For the five variants, our abbreviations of the names, the values of n, k and ω for the five variants, and their categories are shown in Table 4.1. Section 2.2.3 of the Classic McEliece specification [19], states that the public key is an $r \times k$ binary matrix. Table 4.1 shows $\log_2 P$, where $P = r \cdot k$ is the size of the public key in bits. An ISD instance is given by $(\mathbf{H}_0, \mathbf{s}_0, \omega)$. So M_{mat} bits are required to store $\mathbf{H}_0$ and $\mathbf{s}_0$. The values of $\log_2 M_{\text{mat}}$ for the variants of the Classic McEliece are shown in Table 4.1. For Algorithm 4, instead of using the maximum possible ranges of values of parameters given by (4.1), we have restricted to the following ranges of values of the parameters. The concrete results obtained using these choices indicate that there are no interesting TMTO points for values of parameters outside

these ranges.

$$0 \leq \ell \leq 100, \qquad 2 \leq p \leq 30, \qquad -\ell \leq \lambda \leq \ell, \qquad \delta = 1 - 0.025 \cdot i,\ i = 0, \ldots, 12. \tag{5.3}$$

The total number of choices of values for the parameters ℓ, p, λ and δ in the above ranges is 3712800. We compute the sets of effective TMTO points for the variants of Classic McEliece corresponding to Algorithm 4 for both the cases where memory access cost is taken to be constant and the logarithmic memory access cost model is assumed. We have also computed similar sets of effective TMTO points corresponding to Stern's (i.e. with $\delta = 1$ and $\lambda = 0$) and Dumer's (i.e. with $\delta = 1$ and $\lambda = \ell$) algorithms for the ranges of ℓ and p given by (5.3). For every case, the sizes of the sets of effective TMTO points corresponding to Stern's and Dumer's are the same and the corresponding time complexities are equivalent. The memory complexities are also mostly equivalent, though in a few cases they vary a little. We have similarly computed the sets of effective TMTO points corresponding to the Bernstein and Chou's (i.e. with $\delta = 1$) algorithm for the ranges of ℓ, p and λ given by (5.3). Again the sizes of the sets of effective TMTO points are the same as those obtained for Stern's algorithm and the corresponding time complexities are equivalent, while the memory complexities are mostly equivalent and vary a little for a small number of cases.

Remark 7. *The generalisation of Algorithm 4 over Stern's algorithm arises from the use of two parameter λ and δ. From the above discussion, we see that it is the parameter δ which provides the non-triviality of the generalisation. If we set $\delta = 1$ (obtaining Bernstein and Chou's algorithm* isd1*), then we do not obtain any interesting TMTO points beyond what is achieved by Stern's algorithm. It is necessary to allow δ to be less than 1 to explore the TMTO landscape not covered by Stern's algorithm.*

The sets of effective TMTO points for the variants of Classic McEliece obtained from Algorithm 4 are shown in Tables 5.2 and 5.3. The tables also provide the values of the parameters which achieve the corresponding TMTO points. In these tables, rows marked with (*) indicate that Stern's algorithm achieves TMTO points which are equivalent to the corresponding TMTO points achieved by Algorithm 4. Further, Stern's algorithm does not achieve any TMTO point whose time complexity is equivalent to the time complexity of any row not marked with (*). The values in the tables show that the maximum and minimum values of $\lceil \log_2 T \rceil$ remain the same for Algorithm 4 and Stern's algorithm. For Algorithm 4, $\lceil \log_2 T \rceil$ achieves every value between the maximum and the minimum, while for Stern's algorithm only about half of these values are achieved. So Algorithm 4 provides a finer time/memory trade-off compared to Stern's algorithm. Note that for Algorithm 4 sets of 3712800 TMTO points reduce to sets of 6 to 13 effective TMTO points. This underlines the usefulness of the notion of a set of effective TMTO points.

The minimum memory TMTO points for the five Classic McEliece variants have $\log_2 M$ to be equal to 21.45, 22.54, 23.49, 23.45 and 23.79. Comparing with the values of $\log_2 M_{\text{mat}}$ in Table 4.1, one may observe that the minimum memory is marginally greater than $\log_2 M_{\text{mat}}$. The time estimates of the minimum memory trade-off points are less than that of the minimum time estimates of Leon's algorithm (whose memory complexity is M_{mat}) given in Table 5.1 by factors of about 8 to 20.

It is interesting to observe that there are sharp drops in memory requirement at only a moderate increase in the time complexity. As an example, if we consider the estimates which take memory access cost to be constant, increasing $\log_2 T$ by the amount indicated below leads to $\log_2 M$ dropping by the stated amount.

`m3488`: $\log_2 T$ from 147.93 to 148.93; $\log_2 M$ from 44.55 to 34.63;
`m4608`: $\log_2 T$ from 189.98 to 190.80; $\log_2 M$ from 46.22 to 37.70;
`m6688`: $\log_2 T$ from 265.00 to 265.95; $\log_2 M$ from 57.97 to 48.89;
`m6960`: $\log_2 T$ from 265.00 to 265.96; $\log_2 M$ from 67.37 to 58.38;
`m8192`: $\log_2 T$ from 300.00 to 300.96; $\log_2 M$ from 77.99 to 68.96.

This shows that allowing the time complexity to increase by factors which are at most 2 result in the memory requirement dropping by factors varying from 2^8 to 2^{10}. In general, one may observe that the drops in the memory requirement are sharper for smaller values of T and become less sharp for larger values of T. Similar observations hold when we consider the estimates which include logarithmic memory access cost.

We consider the implications of the new TMTO points to the classification in NIST categories of the variants of Classic McEliece. Time estimates of Stern's algorithm from [9] have been used in a discussion [57] regarding the classifications of the five variants. A criticism forwarded against these time estimates was that the corresponding memory requirements are much larger than the size of the public key being attacked.

Since we have obtained the sets of effective TMTO points, we may consider points where the memory size is not much larger than the size of the public key.

Let us first consider `m4608`. In this case the gate count requirement is 2^{207} and the size of the public key is about 2^{22} bits. The time estimates of all the TMTO points for this variant given by Tables 5.2a and 5.2b are below the target 2^{207}. The TMTO point with highest of these time estimates is equal to $(2^{197.99}, 2^{22.52})$ when memory access cost is taken to be constant, and is equal to $(2^{202.00}, 2^{22.54})$ when logarithmic memory access cost is considered. So for `m4608`, it is possible to achieve time complexity less than 2^{207} by a factor of about 2^9 (for constant memory access cost) and about 2^5 (for logarithmic memory access cost) while requiring memory which is only about 1.5 times the size of the public key.

Let us now consider `m6688` and `m6960`. The NIST gate count requirement is 2^{272} and the size of the public key is about 2^{22} bits. From Table 5.3a, we see that for `m6688`, Algorithm 4 has the TMTO points $(2^{269.99}, 2^{29.25})$ and $(2^{271.00}, 2^{27.69})$; and for `m6960`, Algorithm 4 has the TMTO points $(2^{270.00}, 2^{29.93})$ and $(2^{270.85}, 2^{28.83})$. All of these time estimates are slightly smaller than 2^{272}. The corresponding memory requirements, while still being larger than the size of the public key, are not too large. On the other hand, if we consider logarithmic memory access cost, then with memory requirement less than 2^{30}, we obtain time complexities $2^{274.97}$ and $2^{274.93}$ respectively, both of which are greater than the NIST requirement of 2^{272} by factors which are less than 8.

In contrast to the above, for both `m3488` and `m8192`, there is no TMTO point in Tables 5.2 and 5.3, even with constant memory access cost, which has time estimate less than the NIST requirement for classification in the respective categories.

Remark 8. *For* `m4608` *and* `m6688`*, the minimum gate count estimates obtained in [18] for* isd1 *(which subsumes Stern's and Dumer's algorithms) are* $2^{198.93}$ *and* $2^{275.41}$ *respectively.*

For m4608*, the count of* $2^{198.93}$ *is lower than the NIST target of* 2^{207} *while for* m6688*, the count of* $2^{275.41}$ *is about* 10 *times the NIST target of* 2^{272}*. The methodology used in [18] for obtaining gate count estimates incorporates cost of memory access though in a manner which is different from the logarithmic memory access cost considered here (see Remark 6). From Tables 5.2b and 5.3b, the minimum bit complexity estimates for* m4608 *and* m6688 *obtained in the present work are not too far from those obtained in [18]. The main difference with the results reported in [18] is that, as discussed above, the bit complexity estimate for* m4608 *falls below the NIST target and the bit complexity estimate for* m6688 *(and also* m6960*) is a little above the NIST target* even *if we restrict the memory requirement to be not too larger than the size of the public key.*

Table 5.2: TMTO points and the corresponding values of the parameters achieved by Algorithm 4 for m3488 and m4608. Stern's algorithm provides equivalent TMTO points for only the rows marked with (*).

(a) Constant memory access cost.

		$(\log_2 T,\ \log_2 M)$	ℓ	p	λ	δ
m3488	(*)	(147.93, 44.55)	36	8	−36	1.000
	(*)	(148.93, 34.63)	27	6	−27	0.950
	(*)	(149.89, 27.15)	18	4	−18	1.000
		(150.97, 25.71)	17	4	−17	0.925
		(151.96, 24.77)	15	4	−14	0.875
		(152.81, 23.90)	14	4	−14	0.825
		(153.99, 22.80)	13	4	−13	0.750
	(*)	(154.98, 21.44)	13	4	−13	0.975
m4608	(*)	(189.98, 46.22)	38	8	−38	1.000
	(*)	(190.88, 37.70)	28	6	−28	0.975
		(191.81, 35.95)	26	6	−26	0.950
	(*)	(192.95, 27.75)	18	4	−18	0.975
		(193.99, 26.76)	16	4	−16	0.925
		(194.85, 25.81)	15	4	−15	0.875
		(195.74, 24.92)	14	4	−14	0.825
		(196.95, 23.81)	13	4	−13	0.750
	(*)	(197.99, 22.52)	7	3	−7	0.975

(b) Logarithmic memory access cost.

		$(\log_2 T_{\mathrm{ma}},\ \log_2 M)$	ℓ	p	λ	δ
m3488	(*)	(154.00, 34.69)	27	6	16	0.950
	(*)	(154.94, 26.67)	18	4	−18	0.975
		(155.81, 25.70)	16	4	−16	0.925
		(156.91, 24.76)	14	4	−14	0.875
		(157.94, 23.50)	13	4	−13	0.800
	(*)	(158.93, 21.45)	8	3	-8	1.000
m4608	(*)	(195.94, 36.70)	29	6	−29	0.975
	(*)	(196.98, 35.95)	26	6	−26	0.950
	(*)	(197.94, 27.75)	17	4	−17	0.975
		(198.73, 26.76)	16	4	−16	0.925
		(199.92, 25.80)	14	4	−14	0.875
		(200.97, 24.51)	13	4	−13	0.800
	(*)	(202.00, 22.54)	8	3	−2	1.000

Table 5.3: TMTO points and the corresponding values of the parameters achieved by Algorithm 4 for `m6688`, `m6960` and `m8192`. Stern's algorithm provides equivalent TMTO points for only the rows marked with (*).

(a) Constant memory access cost.

		$(\log_2 T,\ \log_2 M)$	ℓ	p	λ	δ
`m6688`	(*)	(265.00, 57.97)	50	10	−16	1.000
	(*)	(265.95, 48.89)	39	8	−39	1.000
	(*)	(267.00, 38.82)	31	6	30	0.975
		(268.80, 36.41)	28	6	−28	0.900
	(*)	(269.99, 29.25)	18	4	−18	0.975
		(271.00, 27.69)	18	4	7	0.900
		(271.93, 26.68)	17	4	−17	0.850
		(273.00, 25.76)	15	4	15	0.800
		(273.95, 24.95)	14	4	−14	0.750
	(*)	(274.89, 23.48)	9	3	−9	0.975
`m6960`	(*)	(265.00, 67.37)	59	12	19	1.000
	(*)	(265.96, 58.38)	48	10	−48	1.000
	(*)	(267.00, 48.29)	39	8	29	0.975
	(*)	(267.91, 39.00)	30	6	−30	0.975
		(268.92, 37.41)	29	6	−29	0.925
	(*)	(270.00, 29.93)	20	4	20	1.000
		(270.85, 28.83)	19	4	−19	0.950
		(271.96, 27.77)	17	4	−17	0.900
		(272.87, 26.75)	16	4	−16	0.850
		(273.80, 25.80)	15	4	−15	0.800
		(274.74, 24.98)	14	4	−14	0.750
	(*)	(275.87, 23.45)	8	3	−8	1.000
`m8192`	(*)	(300.00, 77.99)	68	14	16	1.000
	(*)	(300.96, 68.96)	58	12	-58	1.000
	(*)	(301.90, 58.55)	50	10	−50	0.975
	(*)	(302.90, 49.38)	40	8	−40	0.975
	(*)	(303.96, 39.88)	31	6	−31	0.975
		(305.00, 38.26)	30	6	−30	0.925
		(305.95, 37.44)	28	6	−28	0.900
	(*)	(306.92, 29.99)	19	4	−19	0.975
		(307.80, 28.89)	18	4	−18	0.925
		(308.72, 27.81)	17	4	−17	0.875
		(309.66, 26.79)	16	4	−16	0.825
		(310.96, 25.85)	14	4	−14	0.775
	(*)	(311.85, 23.79)	10	3	−10	1.000

(b) Logarithmic memory access cost.

		$(\log_2 T_{\rm ma},\ \log_2 M)$	ℓ	p	λ	δ
`m6688`	(*)	(270.96, 57.91)	49	10	−49	1.000
	(*)	(271.99, 39.55)	31	6	−31	1.000
		(272.96, 37.98)	29	6	−29	0.950
		(273.99, 36.41)	28	6	−28	0.900
	(*)	(274.97, 28.72)	19	4	−19	0.950
		(275.97, 27.68)	17	4	−17	0.900
		(276.83, 26.67)	16	4	−16	0.850
		(278.00, 25.75)	14	4	−7	0.800
		(278.88, 24.95)	13	4	−13	0.750
	(*)	(279.86, 23.45)	8	3	−8	0.950
`m6960`	(*)	(271.00, 75.99)	68	14	−4	1.000
	(*)	(272.00, 49.33)	40	8	32	1.000
	(*)	(272.94, 39.79)	30	6	−30	1.000
		(274.00, 37.45)	30	6	13	0.925
	(*)	(274.93, 29.91)	20	4	−20	1.000
		(275.91, 28.83)	18	4	−18	0.950
		(276.75, 27.77)	17	4	−17	0.900
		(277.97, 26.75)	15	4	−15	0.850
		(278.84, 25.80)	14	4	−14	0.800
	(*)	(279.89, 23.45)	10	3	−10	1.000
`m8192`	(*)	(306.00, 86.78)	77	16	46	1.000
	(*)	(307.00, 59.88)	52	10	−9	1.000
	(*)	(307.93, 50.43)	41	8	−41	1.000
	(*)	(309.00, 40.70)	31	6	−31	1.000
		(309.97, 39.88)	29	6	−29	0.975
		(310.90, 37.45)	29	6	−29	0.900
	(*)	(311.82, 29.99)	19	4	−19	0.975
		(312.95, 28.35)	18	4	−18	0.900
		(313.90, 27.81)	16	4	−16	0.875
		(314.77, 26.78)	15	4	−15	0.825
		(315.90, 25.44)	14	4	−14	0.750
	(*)	(316.84, 23.78)	9	3	−9	0.975

Previous estimates.

Table 5.4: Previous estimates of the expected number of bit operations and the corresponding memory for Stern's/Dumer's algorithm. The minimum time estimates required by Algorithm 4 are also provided.

name	[9] $(\log_2 T_1, \log_2 M_1)$	[18] $\log_2 T_2$	Algo 4 $(\log_2 T, \log_2 M)$	Algo 4 $(\log_2 T_{\rm ma}, \log_2 M)$
`m3488`	(152.51, 34.68)	156.96	(147.93, 44.55)	(154.00, 34.69)
`m4608`	(194.36, 35.66)	198.93	(189.98, 46.22)	(195.94, 36.70)
`m6688`	(270.46, 37.48)	275.41	(265.00, 57.97)	(270.96, 57.91)
`m6960`	(271.18, 47.58)	-	(265.00, 67.37)	(271.00, 75.99)
`m8192`	(306.63, 67.64)	311.44	(300.00, 77.99)	(306.00, 86.78)

Previous works on concrete estimates of the performance of ISD algorithms have focused on the minimum time that can be achieved for a fixed set of values of n, k and ω, where the minimum is over the appropriate choices of the parameters.

For Classic McEliece, previous estimates of Stern's algorithm in [9, 18] are shown in Table 5.4. To make the comparison clear, we have also included the minimum time estimates and the corresponding memory estimates for Algorithm 4 for both constant and logarithmic memory access costs; equivalent TMTO points can also be obtained from Stern's algorithm (see Tables 5.2 and 5.3). In the column headed by [9], T_1 is the minimum time estimate for Stern's algorithm given in that work and M_1 is the memory required to achieve the corresponding time estimate. In the column headed by [18], T_2 is the minimum time estimate of either Stern's or Dumer's algorithm given in that work, the specific algorithm for each time estimate is not mentioned in [18]. Also, [18] does not provide the memory estimates. The time estimates in [18] include the cost of memory access (see, however, Remark 6), while those in [9] do not[1]. From Table 5.4, we note that the estimates T_1 in [9] are higher than the estimates T achieved by Algorithm 4. The reason is that the techniques of Chase's sequence and early abort were not used in the estimates in [9]. On the other hand, the estimates in [18] are higher than T_{ma}. The values of $\log_2 T_2 - \log_2 T_{\text{ma}}$ for `m3488`, `m4608`, `m6688` and `m8192` are 2.96, 2.99, 4.45 and 5.44 respectively. So the time estimates obtained from our simple cost model are fairly accurate; in particular they are within 1.5% to 1.9% of the time estimates obtained from the sophisticated methodology used in [18].

5.3 Code Used To Obtain Effective Set

We have used the following Python script to compute the effective set of TMTO points.

```
from math import inf, ceil, log2, comb
import sys

def getTuple_Dumer(n,k,w,l,p,lam,delta):
    p1=p//2
    p2=int(ceil(p/2))
    k1=(k+lam)//2
    k2=int(ceil((k+lam)/2))
    L1 = int(ceil((comb(k1,p1))**delta))
    L2 = int(comb(k2,p2))
    PS = (L1 * L2 * comb(n-k-l,w-p))/ comb(n,w)
    TLA = (k**2*(n-k)*(n-k-1)*(3*n-k)) / (4*n**2)
    TSR = (L1+L2)*(2*l+4*(w-p+1)) + L2*(L1/(2**l))*(2*(w-p+1))
    Mmat = (n-k)*(n+1)
```

[1]While [9] mentions the logarithmic memory access cost model, to the best of our understanding Table 4 of [9] provides estimates of bit operations assuming constant memory access time. The values in Table 5.4 in the column headed by [9] are from Table 4 of [9].

```
    Mlst = L1*(l+2*(w-p+1)+p1*log2(k1))
    M = Mmat + Mlst;
    logM = log2(M)
    Titer = log2(TLA+TSR) # Titer = TLA + TSR
    logT = Titer - log2(PS)
    tup = [ceil(logT),ceil(logM),l,p,lam,delta,logT,logM]
    return tup

def FilterTime(resLst): # first filtering
    tresLst = [];
    tresLst.append(resLst[0]);
    val = resLst[0][0]
    for tup in resLst:
        if (tup[0] != val):
            tresLst.append(tup)
            val = tup[0]
    return tresLst

def FilterMemory(resLst): # second filtering and pruning
    tresLst = [];
    tresLst.append(resLst[0]);
    val = resLst[0][1]
    for tup in resLst:
        if (tup[1] > val):
            return tresLst
        if (tup[1] != val):
            tresLst.append(tup)
            val = tup[1]
    return tresLst
# start of main routine: takes three command line parameters: n,k,w

n=int(sys.argv[1]);
k=int(sys.argv[2]);
w=int(sys.argv[3])
lmax = 100;
pmin = 2;
pmax = 30
resLst = []
resLst1 = []

for i in range(0,13):
    delta = 1.0-float(i)*0.025
    for l in range(1,lmax):
        for lam in range(-l,l+1):
```

```
            for p in range(1,pmax):
                tup = getTuple_Dumer(n,k,w,l,p,lam,delta)
                resLst.append(tup)

resLst.sort();
tresLst = FilterTime(resLst);
tresLst = FilterMemory(tresLst)
for tup in tresLst:
    print(tup)
```

5.4 Summary

We have introduced the notion of a set of effective TMTO points. We have shown how to compute such a set for any ISD algorithm. An effective set helps to capture the landscape of time/memory trade-off points of an ISD algorithm uniquely. The sets of effective TMTO points for the new ISD algorithm corresponding to the variants of the Classic McEliece cryptosystems have been obtained and their significance have been discussed in details.

Chapter 6

Conclusion and Future Research Possibilities

In this part of the thesis we have studied Information Set Decoding algorithms with our focus on low weight binary error vector. In Chapter 4 we have introduced a generalisation of Stern's algorithm. This generalisation has been obtained through the use of two new parameters: $\lambda \in [-\ell, \ell]$ and $\delta \in (0, 1]$.

This work has focused on obtaining concrete estimates using the generalised Stern's algorithm. From a more theoretical point of view, it would be of interest to perform a comprehensive asymptotic analysis of the generalised Stern's algorithm following the methodology used in [38]. A more compact asymptotic analysis along the lines of [78, 10] or the more recent [53] would also be of interest.

We have considered the five variants of Classic McEliece to obtain concrete time and memory estimates of this generalisation. We have observed that the generalised Stern's algorithm achieves equivalent TMTO points as that for Stern's algorithm but the values of λ and δ are non-trivial for a number of instances. The work in Chapter 5 shows that an increased number of time/memory trade-off points are achievable due to the new parameter set. The time/memory trade-off of MMT has been proposed by Esser et al. in [53]. Reduced time complexity estimates have been reported for all the five variants of Classic McEliece for three types of memory access costs. For the cube-root memory access cost memory estimates corresponding to the optimal time estimate for each of the five variants of Classic McEliece do not exceed 2^{60} bits. Otherwise different optimal time estimates have been reported for different memory settings. It will be interesting to study the effective set of TMTO points for the MMT version of [53].

Part II

Studies on Polynomial-based hash functions

Chapter 7

Preliminaries

A hash function $H(\cdot)$ maps its inputs from a large domain to short and fixed length members of its range which is smaller in size compared to its domain. Universal hash functions are used to build efficient lookup tables (please refer to [36]). In Cryptography two main types of hash functions are considered: universal hash functions and collision resistant hash functions. These are two different primitives for building a Message Authentication Code. Universal hash functions are relevant to this thesis. We restate some basic concepts and definitions required to understand the concept and application of universal hash functions. Moreover, we explain some basic tools for evaluation of polynomial-based universal hash functions.

7.1 Message Authentication Code

It has been mentioned in Chapter 1 that Message Authentication Codes (MAC) are used to achieve message integrity using a secret shared key. We formally define MAC in this chapter. First we present the basic definition and then move on with the details which are relevant to this thesis.

A MAC is defined over three finite non-empty sets- the key space $\mathcal{K}$, the message space $\mathcal{D}$ and the tag space $\mathcal{G}$. It is a triplet of algorithms $(KeyGen,\ TagGen,\ Verify)$.

- The key-generation algorithm $KeyGen(\cdot)$ is a probabilistic polynomial-time algorithm. It takes in the security parameter $1^{\mathtt{n}}$ as the input and outputs the secret key such that $sk \leftarrow KeyGen(1^{\mathtt{n}})$. The key sk is the shared secret key.

- $TagGen:\ \mathcal{K} \times \mathcal{D} \rightarrow \mathcal{G}$.
  The tag generation algorithm $TagGen(\cdot)$ is a probabilistic polynomial-time algorithm that outputs the tag $\mathtt{t}$ such that $\mathtt{t} \leftarrow TagGen(sk, \mathtt{m})$ where $\mathtt{m} \in \mathcal{D}$.

- $Verify:\ \mathcal{K} \times \mathcal{D} \times \mathcal{G} \rightarrow \{accept, reject\}$.
  The verification algorithm is a deterministic algorithm which takes in as input the secret key sk, the message $\mathtt{m}$ and the tag $\mathtt{t}$. $Verify(sk, \mathtt{m}, \mathtt{t})$ outputs *accept* if $(\mathtt{m}, \mathtt{t})$ is a valid message-tag pair.

Depending on the type of construction, a MAC may need a nonce-space, another finite non-empty set, for a complete definition. The definition of the $TagGen$ and $Verify$ algorithms change to the following:

$TagGen: \mathcal{K} \times \mathcal{D} \times \mathcal{N} \rightarrow \mathcal{G}$
$Verify: \mathcal{K} \times \mathcal{D} \times \mathcal{G} \times \mathcal{N} \rightarrow \{accept, reject\}$.

7.2 Universal Hash Function

Let $\mathcal{D}$ be a non-empty set and $(\mathcal{R}, +)$ be a finite group such that $\#\mathcal{D} > \#\mathcal{R}$ and $\mathcal{K}$ be a finite non-empty set. Let $\{\mathsf{Hash}_\tau\}_{\tau \in \mathcal{K}}$ be a family of functions, such that for each $\tau \in \mathcal{K}$, $\mathsf{Hash}_\tau : \mathcal{D} \rightarrow \mathcal{R}$. The sets $\mathcal{D}$, $\mathcal{K}$ and $\mathcal{R}$ are called the message, key and tag (or digest) spaces respectively. For uniform random choice of τ from $\mathcal{K}$, the following two notions are important for defining security of an universal hash function.

Collision probability: For distinct $x, x' \in \mathcal{D}$, the collision probability of $\{\mathsf{Hash}_\tau\}_{\tau \in \mathcal{K}}$ for the pair (x, x') is defined to be $\Pr_\tau[\mathsf{Hash}_\tau(x) = \mathsf{Hash}_\tau(x')]$.

Differential probability: For distinct $x, x' \in \mathcal{D}$ and any $y \in \mathcal{R}$, the differential probability of $\{\mathsf{Hash}_\tau\}_{\tau \in \mathcal{K}}$ for the triplet (x, x', y) is defined to be $\Pr_\tau[\mathsf{Hash}_\tau(x) - \mathsf{Hash}_\tau(x') = y]$.

Universal Hash Function The family $\{\mathsf{Hash}_\tau\}_{\tau \in \mathcal{K}}$ is said to be ϵ-almost universal (ϵ-AU) if for all distinct x, x' in $\mathcal{D}$, the collision probability for the pair (x, x') is at most ϵ. The family $\{\mathsf{Hash}_\tau\}_{\tau \in \mathcal{K}}$ is said to be ϵ-almost XOR universal (ϵ-AXU) if for all distinct x, x' in $\mathcal{D}$ and any $y \in \mathcal{R}$, the differential probability for the triplet (x, x', y) is at most ϵ.

7.2.1 Polynomial Hashing

Let us denote a finite field by $\mathbb{F}$. Let $m_1, m_2, \ldots, m_\ell$ be ℓ elements of $\mathbb{F}$. For $\ell \geq 0$, a degree-$(\ell - 1)$ polynomial may be formed such that its coefficients are $m_1, \ldots, m_\ell$. The polynomial may be defined as follows:

$$\mathsf{f}(m_1, \ldots, m_\ell) = m_1 \tau^{\ell-1} + m_2 \tau^{\ell-2} + \cdots + m_\ell$$

where the inderterminate $\tau \in \mathcal{K}$.

Horner based polynomial evaluation The above form of polynomial has been heavily used in the literature as discussed in Chapter 8. This polynomial is evaluated at the point τ by $\ell - 1$ multiplications and $\ell - 1$ additions using Horner's rule. To restate the Horner's rule the polynomial f is redefined as $\mathsf{Horner}_\tau(m_1, m_2, \cdots, m_\ell)$. For $\ell \geq 0$, the polynomial $\mathsf{Horner}_\tau(m_1, m_2, \cdots, m_\ell)$ in the variable τ with $m_1, \ldots, m_\ell \in \mathbb{F}$ is defined as follows:

If $\ell = 0$, then $\mathsf{Horner}_\tau() = 0$; and for $\ell > 0$,

$$\left.\begin{array}{rcl} & & \mathsf{Horner}_\tau(M_1, M_2, \cdots, M_\ell) \\ &=& M_1\tau^{\ell-1} + M_2\tau^{\ell-2} + \cdots + M_{\ell-1}\tau + M_\ell \\ &=& (((M_1\tau + M_2)\tau + M_3)\tau + \cdots + M_{\ell-1})\tau + M_\ell. \end{array}\right\} \tag{7.1}$$

By the above definition, computation of Horner on ℓ field elements requires $\ell - 1$ additions and $\ell - 1$ multiplications. It is well known that $\{\mathsf{Horner}_\tau\}_{\tau\in\mathbb{F}}$, is $((\ell-1)/\#\mathbb{F})$-AU. Further, the hash function $\{\tau\mathsf{Horner}_\tau\}_{\tau\in\mathbb{F}}$ is $(\ell/\#\mathbb{F})$-AXU.

BRW Polynomial Bernstein defined a family of polynomials in [17] based on a previous work [92] by Rabin and Winograd. Later this new family of polynomials has been called the BRW polynomials in [95]. For $\ell \geq 0$, $\mathsf{BRW}_\tau(M_1, M_2, \cdots, M_\ell)$ with $M_1, \ldots, M_\ell \in \mathbb{F}$ is a polynomial in the variable τ and is defined as follows:

- $\mathsf{BRW}_\tau() = 0$;
- $\mathsf{BRW}_\tau(M_1) = M_1$;
- $\mathsf{BRW}_\tau(M_1, M_2) = M_1\tau + M_2$;
- $\mathsf{BRW}_\tau(M_1, M_2, M_3) = (\tau + M_1)(\tau^2 + M_2) + M_3$;
- $\mathsf{BRW}_\tau(M_1, M_2, \cdots, M_\ell)$
  $= \mathsf{BRW}_\tau(M_1, \cdots, M_{t-1})(\tau^t + M_t) + \mathsf{BRW}_\tau(M_{t+1}, \cdots, M_\ell)$;
  if $t \in \{4, 8, 16, 32, \cdots\}$ and $t \leq \ell < 2t$.

Let us suppose $\ell \geq 3$. Following Bernstein's work in [17], $\mathsf{BRW}_\tau(M_1, \ldots, M_\ell)$ can be evaluated using $\lfloor \ell/2 \rfloor$ multiplications and $\lfloor \lg \ell \rfloor$ additional squarings to compute $\tau^2, \tau^4, \ldots$.

7.3 Some Topics On Implementation

Saturated and Unsaturated Limb Representation For the prime order field $\mathbb{Z}_p$ all the arithmetic operations are done modulo the prime p. Saturated limb and unsaturated limb representations [85] are two techniques which may be adopted to represent any element $e \in \mathbb{Z}_p$ at the low-level or hardware level. Let each element of $\mathbb{Z}_p$ be represented as b-bit long strings. Let each register be θ-bit wide. Let ω and θ' be two positive integer such that $\omega = \lceil b/\theta' \rceil$. The element e may be represented as

$$e = e_0 + 2^{\theta'} \cdot e_1 + 2^{2\cdot\theta'} \cdot e_2 + \cdots + 2^{(\omega-1)\cdot\theta'} \cdot e_{\omega-1},$$

where each e_i is θ' bit wide, $\forall i \in \{0, \cdots, \omega - 1\}$ but each e_i is stored in a θ-bit wide register. The above representation is called saturated limb representation if θ and θ' are equal and it is called unsaturated if $\theta' < \theta$.

Relevant Instructions One of the algorithms relevant to this thesis which dominates the total number of CPU cycles is the field multiplication algorithm. This thesis requires both sequential and vectorized multiplications. The sequential multiplications are 64-bit operations while the vector/SIMD multiplication instructions are 32-bit operations. The other

field operations include addition and reduction. For the sequential multiplication routines the instructions used heavily are mulx, adcx, adox. The other instructions of this routine includes addq, adcq, shld, shrq and imul. The three instructions mulx, adcx and adox together help in sequential 64-bit multiplications in an interleaved way. This combination makes it possible to maintain two independent carry-chains. The description of the instructions are listed as follows:

- mulx This multiplies two 64-bit unsigned integers and affects neither the carry flag nor the overflow flag.
- adcx This adds two 64-bit unsigned integers along with the carry flag. It affects only the carry flag but leaves the overflow flag unchanged.
- adox This adds two 64-bit unsigned integers along with the overflow flag. It changes only the overflow flag but leaves the carry flag unchanged.
- imul Signed multiplication
- mulq Unsigned 64-bit multiplication. This affects both carry flag and overflow flag.
- addq This adds two 64-bit unsigned integers. It affects the carry flag and overflow flag.
- adcq This adds two 64-bit unsigned integers along with the carry flag. It affects the carry flag and overflow flag.
- shld count, op1, op2: Bitwise shifts left the bits of operand 2 through operand 1 and the count number of most significant bits of operand 1 fill the count number of least significant bits of operand 2.
- shrd count, op1, op2: Bitwise shifts right the bits of operand 2 through operand 1 and the count number of least significant bits of operand 1 fill the count number of most significant bits of operand 2.

The SIMD field multiplication routines use Intel's AVX2 Intrinsics. The AVX2 instruction set gives 4-way vector instructions. The major instructions include the 32-bit multiplication instruction $\_mm256\_mul\_epu32$, the 32-bit addition instruction $\_mm256\_add\_epi32$ and the 64-bit addition instruction $\_mm256\_add\_epi64$. The description of the instruction are listed as follows:

- $\_mm256\_mul\_epu32(\_m256i, \_m256i)$ This takes two 256-bit inputs packed as 64 bits and multiples the low unsigned 32-bit integers from each packed 64-bit elements and store the unsigned 64-bit results in a destination.
- $\_mm256\_add\_epi32(\_m256i, \_m256i)$ Add packed 32-bit integers in x and y, and store the results in a destination.
- $\_mm256\_add\_epi64(\_m256i, \_m256i)$ Add packed 64-bit integers in x and y, and store the results in destination.

Chapter 8

A Brief Survey of Relevant Literature

In this chapter we give a brief survey of the background literature on universal hashing relevant to this thesis. Universal hash functions are important primitives for building MACs. We present an overview focused on the development of the literature of universal hash function starting with the seminal work by Wegman and Carter [74] to the new polynomial-based constructions defined over various types of finite fields. An overview of the said evolution of the literature is presented in section 8.1.

The efficiency of subroutines used for evaluation of polynomials with coefficients drawn from a finite field $\mathbb{F}$ at any point of $\mathbb{F}$ is important for the performance of MACs constructed using polynomial-based hash functions as the underlying primitive. The choice of the finite-field itself is important. Works that cover the said concerns have been briefly covered in sections 8.1 and 8.2.

Surveys on universal hash functions have been done in [40, 14, 95]. The work in [40] gives a detailed discussion of universal hash functions based on polynomials defined over prime order fields.

8.1 Universal Hash Function

The concept of universal hash function has been introduced by Wegman and Carter in [74] where they have also proposed three universal hash functions. They have specified the concept of $universal_2$ family of hash functions. Later the authors have further studied a method [75] of constructing MAC using a universal family of hash functions and a pseudorandom family of functions. This construction is well-known as the Wegman-Carter MAC. The Wegman-Carter type of hash functions are defined over finite-fields and the MAC is built using a family of universal hash function along with a family of pseudorandom functions $F : \mathcal{K} \times \mathcal{N} \rightarrow \mathcal{R}$. The output of the hash function $H_\tau(\cdot)$ is obtained as $H_\tau(\mathtt{m})$ which is then bit-wise xored with $F(k, N)$ where $F(\cdot)$ is a pseudorandom function. In this work the authors have also defined the concept of $universal_n$ family of hash functions.

In [71] Krawczyk et al. have introduced the concept of Almost XOR Universal(AXU) family of hash functions. Besides this concept, it also studied LSFR based hash functions. Strongly universal hash functions have been also studied by Stinson in [5] where several

compositions of two strongly universal hash functions have been considered. The work in [104] by Bierbrauer et al. has shown the relation between error-correcting codes and authentication codes.

Polynomials have been widely used to build universal hash functions. Both multivariate and univariate polynomials have been considered. Multivariate polynomials have been used in MMH [63], NMH* [63], UMAC [72], VMAC etc. MMH [63] and NMH* [63] need the key and message lengths to be equal while UMAC and VMAC need length of the key to be longer than the message length. So for longer messages these constraints form overheads. The key lengths are independent of message length in case of univariate polynomials. Examples of such constructions are PolyR [73], Poly1305 [16], the construction by Bierbrauer et al. [104], the constructions presented in [99] by Shoup, GCM [47], the construction by Bert den Boer [42], the construction by Taylor [103]. The three independent works in [42, 104, 103] put forward the idea to partition the input string into ℓ distinct blocks which are considered to be elements of the field and let these ℓ blocks be the coefficients of the polynomial. Poly1305 adopts this approach. Shoup [99] has described three methods of defining ϵ-AXU hash functions using binary extension field arithmetic: evaluation-type hash, division hash and general division hash. The evaluation type hash interprets the message as the coefficients and the key as the indeterminate of a polynomial $m(x)$ where the hash value is $m(k) \cdot k$ modulo an irreducible polynomial of degree equal to the number of bits per block of the message. By this description, the hash functions studied in [104, 103, 42], UMAC [72], HASH [47, 61], BRW [14], Hash2L [32] are constructed using polynomial evaluation-based hash functions. PolyR [73], Poly1305 [16] and the constructions in [40] are evaluation-based hash functions defined over prime order fields $\mathbb{Z}_p$ for various prime numbers.

The polynomial evaluation based hash function PolyR [73] is defined over prime order field $\mathbb{Z}_{2^{64}-59}$. The key is of size 28 bytes. This work focuses on small length inputs and requires no pre-processing using the key. The performance of PolyR is faster on small length inputs. The performance degrades with the increasing message length.

Another class of universal hash function is the BRW polynomials [14] proposed by Bernstein. This work is based on a previous work by Rabin and Winograd [92]. This needs $\lfloor \ell/2 \rfloor$ number of field multiplications and $\lfloor \log \ell \rfloor$ squarings for ℓ number of blocks. Though the total number of multiplications is significantly lesser than Horner-based evaluation these polynomials are recursive in nature and thus evaluation of these polynomials has some overhead for arbitrarily long messages. This work also proposes a MAC constructed using BRW polynomials. A non-recursive implementation for the evaluation of BRW polynomials has been proposed in [59]. This work has considered binary extension fields.

A two-level universal hash function has been proposed in [32] which uses BRW evaluation in the first layer followed by Horner evaluation in the second layer. The input message is divided into a number of chunks where each chunk consists of some fixed number of message blocks. The number of blocks per chunk is decided so that the recursive overhead may be avoided and the output can be obtained using lesser than ℓ number of multiplications. This work has been defined for $\mathbb{F}_{2^{128}}$ and $\mathbb{F}_{2^{256}}$ using binary extension field arithmetic where $\mathbb{F}_{2^{128}}$ and $\mathbb{F}_{2^{256}}$ are finite fields of 2^{128} and 2^{256} elements respectively and uses pclmulqdq instruction.

A widely deployed Wegman-Carter type MAC which uses evaluation-type hash function but defined over prime-order field is the Poly1305-AES [16]. It is defined over $\mathbb{Z}_{2^{130}-5}$.

Poly1305-ChaCha20 is used in TLS and SSH. Poly1305 divides the message into some ℓ number of blocks each of which is of size 16 bytes. After padding each of the blocks is of size 17 bytes. The padding rule for the last block is different and the final size of the last block depends on the message length. These message blocks are members of $\mathbb{Z}_{2^{130}-5}$. An univariate polynomial of degree ℓ is formed over this prime order field using the padded blocks as coefficients. This polynomial may be evaluated using the usual sequential Horner's method as well as a vectorized interpretation of the Horner's rule. The efficient SIMD implementations are done using vector instructions of the modern micro-architectures like Haswell, Skylake and Kabylake. Originally, the author had proposed an efficient floating-point implementation. Later there have been implementations [60, 24] which have exploited the structure of the prime number and reported significant speed-ups. These works do not use floating-point operations.

GHASH [47] is a polynomial evaluation-based strong universal hash function used in the GMAC authentication code. This is defined over the finite field $\mathbb{F}_{2^{128}}$ and follows binary extension field arithmetic. GMAC is one of the mathematical components of GCM which is one of the modes of operation for block cipher. GCM has been standardized by the NIST. POLYVAL, another universal hash function over $\mathbb{F}_{2^{128}}$, works similar to GHASH but it works on byte-swapped message. Highly optimized implementation of GHASH and POLYVAL have been proposed in [61, 62] using Intel's pclmulqdq instruction.

A recent work [40] has proposed polynomial-evaluation based universal hash function over $\mathbb{Z}_{2^{266}-3}$, $\mathbb{Z}_{2^{174}-3}$, $\mathbb{Z}_{2^{150}-3}$, $\mathbb{Z}_{2^{122}-3}$ and $\mathbb{Z}_{2^{116}-3}$. This work searches for alternatives of Poly1305. The prime $2^{266}-3$ is considered to obtain a higher security than $2^{130}-5$. For 64-bit systems saturated limb representation of an element of $\mathbb{Z}_{2^{266}-3}$ requires five limbs while that of $\mathbb{Z}_{2^{130}-5}$ requires three limbs. So field operations in $\mathbb{Z}_{2^{266}-3}$ require to carry out computations for two extra limbs or 128 bits compared to $\mathbb{Z}_{2^{130}-5}$. The primes $\mathbb{Z}_{2^{174}-3}$ and $\mathbb{Z}_{2^{150}-3}$ have been considered to achieve higher security while keeping the performance almost same as that of Poly1305 because all of the primes require three limbs for representing elements of each of the fields. Each of the primes, $\mathbb{Z}_{2^{122}-3}$ and $\mathbb{Z}_{2^{116}-3}$, requires two limbs for representation and thus the field computations are faster but they give security similar to Poly1305.

Hash functions defined over prime-order fields allow efficient field multiplication for polynomial evaluation by highly exploiting the underlying microarchitecture along with advantage of shorter key size of polynomial evaluation based hash functions. So prime order fields are a good choice for instantiating finite-fields for defining hash functions.

8.2 Arithmetic of prime order field

The Systemization of Knowledge work [40] discusses several issues and trade-offs related to field arithmetic.

The types of prime numbers generally considered for designing universal hash functions are Mersenne prime, pseudo-Mersenne prime and generalized Mersenne prime.

- Mersenne prime is of form $2^m - 1$, where m is a prime number.
- Pseudo-Mersenne prime is of form $2^m - c$, c is an odd positive integer.

- Generalized Mersenne prime [100] is expressed as an irreducible polynomial $f(2^m)$ fulfilling other constraints.

The number of limbs required to represent any member of the chosen $\mathbb{Z}_p$ depends on the size of the prime itself. An element e of $\mathbb{Z}_p$ is mapped to a binary string of length $\lceil \log_2 p \rceil$ bits. So larger the prime more is the number of limbs at the low-level. This is true for both saturated and unsaturated representations. Multiplication routines in larger prime order fields are also costlier than a smaller prime order field. Apart from costly field multiplication routines, increased number of limbs also causes frequent loading and unloading of the content(s) of limb(s) to and from the memory respectively. The number of registers available at the low-level is limited and some instructions follow strict register requirements. So carrying out field operations with large number of limbs have such overheads. Another important aspect of the structure of p for saturated representation is the number of empty bit positions in the leftmost or most significant limb. In case of Pseudo-Mersenne primes, the value of c should preferably be small.

The polynomial evaluation based hash functions require field multiplication algorithm which is costlier than the rest of the field operations in terms of number of CPU cycles. Multiplication in a prime order field implies a multiplication followed by a reduction modulo the prime p. So number of multiplication operations required need equal number of reductions modulo p. In an attempt to reduce the number of reductions and thereby the CPU cycles required per field multiplication, two reduction techniques have been be adopted. The reduction may be partial or delayed. A reduction can be delayed up to the point of execution till the underlying register-size allows to add unreduced partial products before any bit is lost. The limb from which the bit is lost depends on the representation. For unsaturated representation bits may be lost from any number of the total number of limbs. For saturated representations, a bit may be lost from the leftmost limb. Delaying reductions requires storing unreduced results in larger number of limbs. Partial reductions, which are naturally less costly than full reduction, may be another option which requires lesser number of limbs to store partially reduced results. A vectorized implementation of Poly1305 with very competitive performance have been reported in [60]. This work uses unsaturated limb representation and partial reduction.

The reduction routine varies on the choice of p. The reduction algorithm for Mersenne primes are simpler than pseudo-Mersenne primes which in turn has simpler reduction algorithm than Generalized Mersenne primes.

Chapter 9

Improved SIMD implementation of Poly1305

Confidentiality and integrity of data flowing through the internet is of paramount importance. The Transport Layer Security (TLS) protocol is the leading security protocol for internet communications. TLS provides a variety of primitives for different cryptographic functionalities. Among the algorithms which are part of TLS is Poly1305 [16] (in combination with ChaCha [12]). Apart from TLS, Poly1305 is part of various other cryptographic libraries: it has been standardised by the IETF and it is part of the NaCl [21] library (in combination with Salsa20 [15]). More concretely, Google uses ChaCha-Poly1305 to secure communication between Chrome browsers on Android and Google websites. The Wikipedia page[1] for Poly1305 provides further details about real-world deployment of Poly1305. Given the widespread use of Poly1305, efficient software implementation of the algorithm is an important issue. Modern processors are moving towards providing vector instructions. These instructions allow single-instruction, multiple-data (SIMD) implementation of a variety of algorithms. The importance of vectorisation in modern processors has been highlighted by Bernstein[2].

The first vectorized implementation of Poly1305 has been proposed by Goll-Gueron [60] in 2015 using Intel Intrinsics. They showed a way to divide the Poly1305 computation into $\mathbf{d}$ independent and parallel computation streams. Concrete implementations were provided in [60] for $\mathbf{d} = 4$ and $\mathbf{d} = 8$ on modern Intel processors. The vectorization is achieved by implementing 4-way Horner's method but the 4-way Horner is not applied throughout the input length. The input must have number of 16-byte blocks in multiples of four. Otherwise the 4-way Horner's evaluation is done for a lesser length and beyond that length sequential evaluation is followed but the multiplications are done using vector instructions. We propose a simple balancing technique to improve the Goll-Gueron strategy so that the 4-way SIMD evaluation is done throughout the length of the input. We have modified the code accompanying the Goll-Gueron paper to obtain an implementation of the 4-way vectorisation strategy using our strategy. For message lengths up to 4000 bytes, this leads to significant speed im-

[1] `https://en.wikipedia.org/wiki/Poly1305`, accessed on 27th June, 2019.

[2] https://groups.google.com/a/list.nist.gov/forum/ #!searchin/pqc-forum/vectorization%7Csort:date/pqc-forum/mmsH4k3j_1g/JfzP1EBuBQAJ, accessed on 27th June, 2019.

provements for messages where the number of blocks is not divisible by four. Comparative speed measurements of the new algorithm and the Goll-Gueron algorithm have been made on the Haswell, Kaby Lake and Skylake processors. For each of these micro-architectures we have considered two compilers, namely GCC and Clang.

This chapter is based on the work in [24].

9.1 Introducing The Balancing Technique

Suppose $\mathbf{d} = 4$. The top level view of the Goll-Gueron algorithm is as follows. The message is formatted into blocks. The algorithm processes 4 blocks at a time. If the number of blocks in the message is a multiple of 4, then the algorithm uniformly processes all the blocks. On the other hand, if the number of blocks is not a multiple of 4, then at the end the parallelism breaks down and the tail of the message consisting of one to three blocks have to be processed separately.

We provide a simple idea to improve the Goll-Gueron algorithm. The Poly1305 algorithm essentially computes a polynomial over a finite field whose coefficients are the blocks of the message. Prepending the message with some zero blocks (i.e., blocks corresponding to the zero element of the field), the output of the Poly1305 algorithm remains unchanged. We take advantage of this feature by prepending the message with one to three zero blocks so that overall the number of blocks in the message is a multiple of 4. Then the processing of the blocks can be done 4 at a time in a uniform manner.

The above strategy opens up a further opportunity for improvement. Suppose that the number of blocks in the message is 1 modulo 4. Then 3 zero blocks would need to be prepended. Consider the initial 4-way multiplication. This consists of 3 zero blocks and one message blocks. So, applying a general 4-way multiplication routine in this case leads to many multiplications by zeros. This is wasteful. We describe a new method to perform such an initial multiplication which is much faster than a general 4-way multiplication. Similarly, we extend this to the case where the number of blocks in the message is 2 modulo 4. In the case where the number of blocks is 3 modulo 4, there is no advantage in trying to reduce the number of multiplications using a new initial multiplication algorithm.

Goll and Gueron [60] also describe a 8-way vectorisation strategy. Our idea of simplifying the parallelism and improving the initial multiplication extends to the 8-way vectorisation. More generally, our algorithmic improvement over the Goll-Gueron strategy applies to all processors which support vector instructions.

9.2 Description of Poly1305 Hash Function

Let $p = 2^{130} - 5$ and $\mathbb{F}_p$ be the finite field of p elements. The Poly1305 hash function maps a message into an element of $\mathbb{F}_p$. In the original description [16] of Poly1305, the message is a sequence of bytes. The later work [60] considered the message to be a sequence of bits; if the

number of bits in the message is a multiple of 8, then the description in [60] coincides with the original description in [16]. Following [60], we provide a description of Poly1305 where a message is a sequence of bits.

Suppose, a message X consists of $L \geq 0$ bits. If $L = 0$, define ℓ to be 0; otherwise, let $\ell \geq 1$ be such that $L = 128(\ell - 1) + r$ where $1 \leq r \leq 128$. Write the message as a concatenation of ℓ strings, i.e., $X = X_0 || \cdots || X_{\ell-1}$ such that $X_0, \ldots, X_{\ell-2}$ each have length 128 bits and $X_{\ell-1}$ has length r bits. For $i = 0, \ldots, \ell - 2$, define $M_i = X_i || 1$, and $M_{\ell-1} = X_{\ell-1} || 10^{128-r}$. This ensures that the length of M_i is 129 bits for $i = 0, \ldots, \ell - 1$. Let $\mathsf{format}(X)$ be the map from a message M to $(M_0, \ldots, M_{\ell-1})$.

From the above description, X_i is the binary representation (written with the least significant bit on the left) of an integer which is less than 2^{129}. For convenience of notation, we will identify the binary string M_i with the integer it represents. Note that the M_i's cannot take all the values in the set $\{0, \ldots, 2^{129} - 1\}$; in particular, none of the M_i's can be zero.

The Poly1305 hash function uses a key R which is an element of $\mathbb{F}_p$. The specification of Poly1305 requires some of the bits R to be set to zero. This was done for efficiency purposes. For the SIMD implementation that we consider, the setting of certain bits of R to be zero does not either help or hamper the efficiency. So, we skip the details of the exact form of R which are given in [16].

The Poly1305 hash function is defined as follows. Given a message M consisting of $L \geq 0$ bits and a key $R \in \mathbb{F}_p$, the output is defined by Eq. (9.1).

$$\mathsf{Poly1305}_R(M) = M_0 R^\ell + M_1 R^{\ell-1} + \cdots + M_{\ell-2} R^2 + M_{\ell-1} R \tag{9.1}$$

where $(M_0, \ldots, M_{\ell-1}) = \mathsf{format}(M)$.

Since the map $\mathsf{format} : M \to (M_0, \ldots, M_{\ell-1})$ is injective, by an abuse of notation, instead of writing $\mathsf{Poly1305}_R(M)$, we will write $\mathsf{Poly1305}_R(M_0, \ldots, M_{\ell-1})$. Note that if M is the empty string, i.e., $L = 0$, then $\ell = 0$ and so $\mathsf{Poly1305}_R(M)$ is 0 (the zero element of $\mathbb{F}_p$).

9.3 Goll-Gueron SIMD Implementation

Given $M_0, \ldots, M_{\ell-1} \in \mathbb{F}_p$ and R, $\mathsf{Poly}(R; M_0, \ldots, M_{\ell-1})$ is defined as follows.

$$\mathsf{Poly}_R(M_0, \ldots, M_{\ell-1}) = M_0 R^{\ell-1} + \cdots + M_{\ell-2} R + M_{\ell-1}. \tag{9.2}$$

So,

$$\mathsf{Poly1305}_R(M_0, \ldots, M_{\ell-1}) = R\ \mathsf{Poly}_R(M_0, \ldots, M_{\ell-1}) \tag{9.3}$$

The definition of Poly in Eq. (9.2) permits the computation of the output using Horner's rule in the following manner.

$$\begin{aligned} &\mathsf{Poly}(R; M_0, \ldots, M_{\ell-1}) \\ &\quad = ((\cdots(((M_0 R + M_1)R) + M_2)R + \cdots)R + M_{\ell-1}. \end{aligned} \tag{9.4}$$

This requires $\ell - 1$ multiplications and $\ell - 1$ additions over $\mathbb{F}_p$. As a result, Poly1305 can be computed using ℓ multiplications and $\ell - 1$ additions over $\mathbb{F}_p$.

Horner's rule is a sequential method of evaluation. One way to exploit parallelism in the computation is to divide the sequence $(M_0, \ldots, M_{\ell-1})$ into $\mathbf{d} \geq 2$ subsequences and apply Horner's rule to each of the subsequence. This allows alternatively performing $\mathbf{d}$ simultaneous multiplications and $\mathbf{d}$ simultaneous additions. Such a strategy has been called $\mathbf{d}$-decimated Horner evaluation [32]. Goll and Gueron [60] described SIMD implementations of Poly1305 based on $\mathbf{d}$-decimated Horner evaluation. They considered two values of $\mathbf{d}$, namely, $\mathbf{d} = 4$ and $\mathbf{d} = 8$ leading to 4-way and 8-way SIMD implementations respectively. We provide details for $\mathbf{d} = 4$, the case of $\mathbf{d} = 8$ being similar.

Let $\rho = \ell \bmod 4$ and $\ell' = (\ell - \rho)/4$. The computation of $\mathsf{Poly1305}_R(M_0, \ldots, M_{\ell-1})$ can be done in the following manner. Let

$$\begin{aligned} P = \quad & R^4\ \mathsf{Poly}(R^4; M_0, M_4, M_8, \ldots, M_{4\ell'-4}) \\ & + R^3\ \mathsf{Poly}(R^4; M_1, M_5, M_9, \ldots, M_{4\ell'-3}) \\ & + R^2\ \mathsf{Poly}(R^4; M_2, M_6, M_{10}, \ldots, M_{4\ell'-2}) \\ & + R\ \mathsf{Poly}(R^4; M_3, M_7, M_{11}, \ldots, M_{4\ell'-1}) \end{aligned} \tag{9.5}$$

Then

$$\begin{aligned} & \mathsf{Poly1305}_R(M_0, \ldots, M_{\ell-1}) \\ & = \begin{cases} P \text{ if } \rho = 0; \\ R\ \mathsf{Poly}(R; P + M_{\ell-\rho}, M_{\ell-\rho+1}, \ldots, M_{\ell-1}) \quad \text{if } \rho > 0. \end{cases} \end{aligned} \tag{9.6}$$

Define

$$\mathbf{R} = \left(R^4, R^3, R^2, R\right)^\intercal; \tag{9.7}$$

$$\mathbf{R}_4 = (R^4, R^4, R^4, R^4)^\intercal; \tag{9.8}$$

$$\mathbf{C}_i = (M_{4i}, M_{4i+1}, M_{4i+2}, M_{4i+3})^\intercal; \text{ for } i = 0, 1, \ldots, \ell' - 1. \tag{9.9}$$

The computation in Eq. (9.6) is described in vector form in Algorithm 6. In the description of Algorithm 6, a temporary vector $\mathbf{T} = (T_0, T_1, T_2, T_3)$ is used and $\circ$ denotes the Hadamard (i.e., component-wise) product of vectors. The quantity P is a temporary field element.

9.3.1 Vector Multiplication

Recall that $p = 2^{130} - 5$ and so any element of $\mathbb{F}_p$ can be represented using 130 bits. Let $\theta = 2^{26}$. An element $X \in \mathbb{F}_p$ can be written in base θ as follows.

$$X = x_0 + x_1\theta + x_2\theta^2 + x_3\theta^3 + x_4\theta^4$$

where $0 \leq x_0, \ldots, x_4 \leq 2^{26} - 1$. Then $(x_4, x_3, x_2, x_1, x_0)$ is called a 5-limb representation of X.

Given 5-limb representations $(x_4, x_3, x_2, x_1, x_0)$ and $(y_4, y_3, y_2, y_1, y_0)$ of X and Y respectively, the product $X \cdot Y \bmod p$ is computed in two steps.

Algorithm 6 Structure of Goll-Gueron 4-way vectorisation of Poly1305 computation. Refer to (Equations 9.7, 9.8 and 9.9) for the definition of the vector quantities.

Input: $(M_0, \ldots, M_{\ell-1})$
Output: $\mathsf{Poly1305}_R(M_0, \ldots, M_{\ell-1})$

1: $\mathbf{T} \leftarrow \mathbf{C}_0$
2: **for** $i = 1$ to $\ell' - 1$ **do**
3: $\quad \mathbf{T} \leftarrow \mathbf{R}_4 \circ \mathbf{T} + \mathbf{C}_{4i}$
4: **end for**
5: $\mathbf{T} \leftarrow \mathbf{R} \circ \mathbf{T}$
6: $P = T_0 + T_1 + T_2 + T_3$
7: **if** $\rho > 0$ **then**
8: $\quad P \leftarrow R\ \mathsf{Poly}(R; P + M_{\ell-\rho}, M_{\ell-\rho+1}, \ldots, M_{\ell-1})$
9: **end if**
10: **return** P

Multiplication step: Let the product of X and Y modulo p be denoted by Z. After the multiplication step $Z = z_0 + z_1\theta + \cdots + z_4\theta^4$ is obtained where $z_0, \ldots, z_4$ are defined as follows.

$$\begin{aligned}
z_0 &= x_0 \cdot y_0 + 5 \cdot x_1 \cdot y_4 + 5 \cdot x_2 \cdot y_3 + 5 \cdot x_3 \cdot y_2 + 5 \cdot x_4 \cdot y_1 \\
z_1 &= x_0 \cdot y_1 + x_1 \cdot y_0 + 5 \cdot x_2 \cdot y_4 + 5 \cdot x_3 \cdot y_3 + 5 \cdot x_4 \cdot y_2 \\
z_2 &= x_0 \cdot y_2 + x_1 \cdot y_1 + x_2 \cdot y_0 + 5 \cdot x_3 \cdot y_4 + 5 \cdot x_4 \cdot y_3 \\
z_3 &= x_0 \cdot y_3 + x_1 \cdot y_2 + x_2 \cdot y_1 + x_3 \cdot y_0 + 5 \cdot x_4 \cdot y_4 \\
z_4 &= x_0 \cdot y_4 + x_1 \cdot y_3 + x_2 \cdot y_2 + x_3 \cdot y_1 + x_4 \cdot y_0.
\end{aligned} \tag{9.10}$$

Note that each z_i is less than 2^{64}.
By $\mathsf{mult}((x_4, \ldots, x_0), (y_4, \ldots, y_0))$ we will denote the vector $(z_0, \ldots, z_4)$.

Reduction step: $W = w_0 + w_1\theta + \cdots + w_4\theta^4$ is obtained such that $W \equiv Z \bmod p$ such that each w_i can be represented using either 26 or 27 bits. By $\mathsf{reduce}(z_4, \ldots, z_0)$ we will denote the vector $(w_4, \ldots, w_0)$. For the details of the reduction step, we refer to [16].

Suppose X is a fixed quantity and the product $X \cdot Y \bmod p$ is required to be computed. Note that the computation in Eq. (9.10) is helped by pre-computing and storing $(5 \cdot x_4, 5 \cdot x_3, 5 \cdot x_2, 5 \cdot x_1)$ along with the 5-limb representation $(x_4, x_3, x_2, x_1, x_0)$ of X.

Vector multiplication Algorithm 6 requires the vector multiplication

$$\mathbf{R}_4 \circ \mathbf{T} = (R^4 \cdot T_0, R^4 \cdot T_1, R^4 \cdot T_2, R^4 \cdot T_3)^\intercal.$$

Note that the multiplication in Step 3 of Algorithm 6 has one of the operands to be fixed to $\mathbf{R}_4$ while the other operand changes. Goll and Gueron [60] presented a very efficient SIMD algorithm to do this multiplication.

Table 9.1: Alignment of R^4 in two 256-bit registers

$\mathtt{r}_0$	$5 \cdot r_4$	$5 \cdot r_3$	$5 \cdot r_2$	r_4	r_3	r_2	r_1	r_0
$\mathtt{r}_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$	$5 \cdot r_1$

Table 9.2: Packing four 256-bit words in three 256-bit registers

$\mathtt{t}_0$	$t_{3,2}$	$t_{3,0}$	$t_{2,2}$	$t_{2,0}$	$t_{1,2}$	$t_{1,0}$	$t_{0,2}$	$t_{0,0}$
$\mathtt{t}_1$	$t_{3,3}$	$t_{3,1}$	$t_{2,3}$	$t_{2,1}$	$t_{1,3}$	$t_{1,1}$	$t_{0,3}$	$t_{0,1}$
$\mathtt{t}_2$	x	$t_{3,4}$	x	$t_{2,4}$	x	$t_{1,4}$	x	$t_{0,4}$

The vector $\mathbf{T} = (T_0, T_1, T_2, T_3)$ has four elements of $\mathbb{F}_p$. Each of these elements has a 5-limb representation. Let $(t_{i,4}, \ldots, t_{i,0})$ be the 5-limb representation of T_i, $i = 0, 1, 2, 3$. So, a total of 20 26-bit quantities are required to store $\mathbf{T}$. Since intermediate results are not fully reduced, some of the $t_{i,j}$'s can be 27-bit quantities. Let $(r_4, \ldots, r_0)$ be the 5-limb representation of R^4. Also, the vector $(5 \cdot r_4, 5 \cdot r_3, 5 \cdot r_2, 5 \cdot r_1)$ is stored.

The 4-way SIMD implementation of Goll and Gueron [60] uses 256-bit words. Each 256-bit word is considered to be 8 32-bit words. So, the 20 26-bit quantities of $\mathbf{T}$ can be stored in three 256-bit words. The vectors $(r_4, \ldots, r_0)$ and $(5 \cdot r_4, 5 \cdot r_3, 5 \cdot r_2, 5 \cdot r_1)$ can be stored in two 256-bit words. The multiplication $\mathbf{W} = \mathbf{R}_4 \circ \mathbf{T}$ consists of two steps.

Vector multiplication step: This step takes as input the three 256-bit words representing $\mathbf{T} = (T_0, T_1, T_2, T_3)$ and the two 256-bit words representing $(r_4, \ldots, r_0)$ and $(5 \cdot r_4, 5 \cdot r_3, 5 \cdot r_2, 5 \cdot r_1)$. It produces as output five 256-bit words $S_0, \ldots, S_4$, where $S_i = (s_{i,3}, s_{i,2}, s_{i,1}, s_{i,0})$ and each $s_{i,j}$ is a 64-bit word. Further, $(s_{0,j}, s_{1,j}, s_{2,j}, s_{3,j}, s_{4,j})$ is $\mathsf{mult}((r_4, \ldots, r_0), (t_{j,4}, \ldots, t_{j,0}))$ for $j = 0, \ldots, 3$. Let $\mathbf{S} = (S_0, \ldots, S_4)$. By $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$ we will denote $\mathbf{S}$.

Vector reduction step: This step takes as input $S_0, \ldots, S_4$ and produces as output three 256-bit words which stores the twenty 26-bit (or 27-bit) words of the result. Let us call the result as $\mathbf{W}$. So, $\mathbf{W} = (W_0, W_1, W_2, W_3)$. Now let us define $W_j = (w_{j,4}, \ldots, w_{j,0})$, $j = 0, 1, 2, 3$. Then $(w_{j,4}, \ldots, w_{j,0})$ is the output of $\mathsf{reduce}(s_{0,j}, s_{1,j}, s_{2,j}, s_{3,j}, s_{4,j})$. $\mathbf{W}$ will denote $\mathsf{vecReduce}(S)$.

In terms of the above notation, the computation $\mathbf{W} = \mathbf{R}_4 \circ \mathbf{T}$ consists of the following two steps: $\mathbf{S} \leftarrow \mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$; $\mathbf{W} \leftarrow \mathsf{vecReduce}(\mathbf{S})$. Note that $\mathbf{T}$ is stored in three 256-bit words and the output $\mathbf{W}$ is also stored in three 256-bit words. This ensures that the same multiplication algorithm can be applied to multiply $\mathbf{R}_4$ and $\mathbf{W}$, and so on.

We provide the top level schematics of $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$ and $\mathsf{vecReduce}(\mathbf{S})$. The 5-limb representation $(r_4, r_3, r_2, r_1, r_0)$ of R^4 and $(5 \cdot r_4, 5 \cdot r_3, 5 \cdot r_2, 5 \cdot r_1)$ are represented in two 256-bit words as represented in Table 9.1. The vector $\mathbf{T} = (T_0, T_1, T_2, T_3)$ is stored in three 256-bit words as in Table 9.2, where $\mathtt{x}$ denotes an undetermined quantity that is not used in the algorithm. The Intel AVX2 implementation of $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$, uses a number of SIMD permutation operations on $\mathtt{t}_0, \mathtt{t}_1$ and $\mathtt{t}_2$ followed by 32-bit SIMD multiplication operations

Table 9.3: Packing result of $\mathsf{vecMult}(\mathbf{R^4}, \mathbf{T})$ in five 256-bit registers

S_0	$s_{0,3}$	$s_{0,2}$	$s_{0,1}$	$s_{0,0}$
S_1	$s_{1,3}$	$s_{1,2}$	$s_{1,1}$	$s_{1,0}$
S_2	$s_{2,3}$	$s_{2,2}$	$s_{2,1}$	$s_{2,0}$
S_3	$s_{3,3}$	$s_{3,2}$	$s_{3,1}$	$s_{3,0}$
S_4	$s_{4,3}$	$s_{4,2}$	$s_{4,1}$	$s_{4,0}$

Table 9.4

Name of Intrinsic	Count	(Latency, Throughput(CPI)[3])		
		Skylake	Haswell	Kaby Lake
_mm256_mul_epu32	25	(5,0.5)	(5,1)	–
_mm256_set_epi32	1	–	–	–
_mm256_add_epi64	20	(1,0.33)	(1,0.5)	–
_mm256_permutevar8x32_epi32	9	–	–	–
_mm256_permute4x64_epi64	4	–	–	–

Table 9.5: Packing result of $\mathsf{vecMult}(\mathbf{R^4}, \mathbf{T})$ in three 256-bit registers after reduction modulo $2^{130} - 5$

$\mathtt{w_0}$	$w_{3,2}$	$w_{3,0}$	$w_{2,2}$	$w_{2,0}$	$w_{1,2}$	$w_{1,0}$	$w_{0,2}$	$w_{0,0}$
$\mathtt{w_1}$	$w_{3,3}$	$w_{3,1}$	$w_{2,3}$	$w_{2,1}$	$w_{1,3}$	$w_{1,1}$	$w_{0,3}$	$w_{0,1}$
$\mathtt{w_2}$	x	$w_{3,4}$	x	$w_{2,4}$	x	$w_{1,4}$	x	$w_{0,4}$

with $\mathtt{r_0}$ and $\mathtt{r_1}$ and 64-bit SIMD operations to accumulate the results. Finally, the result of $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$ is $(S_0, \ldots, S_4)$ and is stored as in Table 9.3. The number of different operations (in Intel intrinsics) required by $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{T})$ is given in Table 9.4.

The $\mathsf{vecReduce}(\mathbf{S})$ implementation takes as input the five 256-bit words $S_0, \ldots, S_4$ and produces as output the vector $\mathbf{W} = (W_0, W_1, W_2, W_3)$ stored in three 256-bit words $\mathtt{w_0}, \mathtt{w_1}$ and $\mathtt{w_2}$ as in Table 9.5. The evaluation in Step 8 of Algorithm 6 is also done using vector operations. This is not clearly described in the paper [60] and can be understood from the accompanying code. Since Step 8 is not relevant to our algorithm we do not describe the details of its computation.

9.3.2 Lazy Reduction

Consider the loop in Steps 2 - 4 of Algorithm 6. Step 3 consists of one 4-way field multiplication $\mathbf{R}_4 \circ \mathbf{T}$ followed by a 4-way field addition. As explained above, the operation $\mathbf{R}_4 \circ \mathbf{T}$ can be realised as $\mathsf{vecReduce}(\mathsf{vecMult}(\mathbf{R}_4, \mathbf{T}))$. So, $\mathbf{R}_4 \circ \mathbf{T}$ requires one vecMult and one vecReduce operation. For long messages, it is possible to improve this by using a lazy reduction strategy. Such a strategy consists of performing a series of successive vecMult and 4-way field additions followed by a single reduction. We provide more details.

[3]CPI=Cycles Per Instruction

Steps 2 - 4 has $(\ell' - 1)$ 4-way field multiplications and 4-way field additions. Suppose we bunch these operations into groups where each group has λ 4-way field multiplications and λ 4-way field additions. If λ does not divide $\ell' - 1$, the last group may have lesser number of such operations. With $\mathbf{T}$ initialised to $\mathbf{C}_0$, the computation of the j-th group, $j = 0, \ldots, \lfloor(\ell' - 1)/\lambda\rfloor$, processes $\mathbf{C}_{\lambda j+1}, \ldots, \mathbf{C}_{\lambda j+\lambda}$. The actual computation is the following.

$$\mathbf{T} \leftarrow \mathbf{R}_{4\lambda} \circ \mathbf{T} + \mathbf{R}_{4(\lambda-1)} \circ \mathbf{C}_{\lambda j+1} + \cdots + \mathbf{R}_4 \circ \mathbf{C}_{\lambda j+\lambda-1} + \mathbf{C}_{\lambda j+\lambda}. \tag{9.11}$$

In the above, $\mathbf{R}_{4k} = (R^{4k}, R^{4k}, R^{4k}, R^{4k})^\intercal$ for $k = 1, \ldots, \lambda$. For the multiplication by $\mathbf{R}_{4k}$, the field elements R^{4k} and $5 \cdot R^{4k}$ are precomputed and stored (only the first 4 limbs of $5 \cdot R^{4k}$ are stored).

As written, Eq. (9.11) requires λ 4-way field multiplications and λ 4-way field additions. Note however, that the results of the field multiplications are simply added together. This suggests the following lazy reduction strategy to perform the computation given in Eq. (9.11).

$\mathbf{W}_0 \leftarrow \mathsf{vecMult}(\mathbf{R}_{4\lambda}, \mathbf{T})$;
$\mathbf{W}_k \leftarrow \mathsf{vecMult}(\mathbf{R}_{4k}, \mathbf{C}_{\lambda j+k})$, $k = 1, \ldots, \lambda - 1$;
$\mathbf{B} \leftarrow \mathbf{W}_0 + \cdots + \mathbf{W}_{\lambda-1} + \mathbf{C}_{\lambda j+\lambda}$;
$\mathbf{T} \leftarrow \mathsf{vecReduce}(\mathbf{B})$.

This method requires λ vecMult operations, λ 4-way field addtions and a single vecReduce operation. Compared to a direct computation of Eq. (9.11), the lazy reduction strategy reduces the number of vecReduce operations by roughly a factor of $(\lambda - 1)/\lambda$. This would suggest that using a higher value of λ should always be beneficial. This, however, is not the case. As λ increases, so does the number of pre-computed quantities. The time for pre-computation has to be taken into account. For a higher value of λ not all the pre-computed quantities can be kept in the registers and as a result, the number of load/store operations would increase substantially. Also, adding too many of the products without a reduction can lead to a overfull in the register. These reasons prevent the use of high values of λ. In [60], the lazy reduction strategy was used for messages of lengths at least 832 bytes and with the values of λ to be 2 and 3.

9.4 A New SIMD Implementation of Poly1305

Algorithm 6 implements the computation in Eq. (9.5). If $4|\ell$ (i.e., $\rho = 0$), then the 4-way SIMD computation proceeds uniformly throughout. However, if $4 \not| \ell$, then the 4-way SIMD computation in Algorithm 6 proceeds uniformly for ℓ' steps. Additionally, the computation in Step 8 is required making the computation non-uniform. For short messages, this leads to a significant penalty.

By making a simple modification, it can be ensured that the 4-way SIMD proceeds uniformly throughout. As before, let $\rho = \ell \bmod 4$. If $\rho = 0$, let $m = \ell$; and if $\rho > 0$, let $m = \ell + 4 - \rho$. Given the sequence $(M_0, \ldots, M_{\ell-1})$ obtained as $\mathsf{format}(M)$, define the sequence $(D_0, \ldots, D_{m-1})$ where if $\rho = 0$, then $D_i = M_i$ for $i = 0, \ldots, \ell - 1$ and if $\rho > 0$, then

$$D_i = \begin{cases} 0, \ i = 0, \ldots, 3-\rho; \\ M_{i-4+\rho}, \ i = 4-\rho, \ldots, m-1. \end{cases}$$

In the definition $D_i = 0$, the '0' is the zero element of $\mathbb{F}_p$ and not the bit 0. The zero element of $\mathbb{F}_p$ is represented in binary using a zero block which is a binary string consisting of 129 zero bits. In other words, the sequence $(M_0, \ldots, M_{\ell-1})$ is prepended using a minimum number of zero blocks to make the length a multiple of 4. Since the initial zeros have no effect on the computation of $\mathsf{Poly1305}_R(D_0, \ldots, D_{m-1})$, we have

$$\mathsf{Poly1305}_R(D_0, \ldots, D_{m-1}) = \mathsf{Poly1305}_R(M_0, \ldots, M_{\ell-1}). \tag{9.12}$$

Let us define $m' = m/4$. Then, the computation of $\mathsf{Poly1305}_R(D_0, \ldots, D_{m-1})$ can be written as follows.

$$\begin{aligned} &\mathsf{Poly1305}_R(D_0, \ldots, D_{m-1}) \\ &\quad = \quad R^4 \ \mathsf{Poly}(R^4; D_0, D_4, \ldots, D_{m-4}) \\ &\quad + \ R^3 \ \mathsf{Poly}(R^4; D_1, D_5, \ldots, D_{m-3}) \\ &\quad + \ R^2 \ \mathsf{Poly}(R^4; D_2, D_6, \ldots, D_{m-2}) \\ &\quad + \ R^1 \ \mathsf{Poly}(R^4; D_3, D_7, \ldots, D_{m-1}). \end{aligned} \tag{9.13}$$

In a manner similar to Eq. (9.7, 9.8,and 9.9), define

$$\mathbf{R} = \left(R^4, R^3, R^2, R\right)^\intercal; \tag{9.14}$$

$$\mathbf{R}_4 = (R^4, R^4, R^4, R^4)^\intercal; \tag{9.15}$$

$$\mathbf{D}_i = (D_{4i}, D_{4i+1}, D_{4i+2}, D_{4i+3})^\intercal; \text{ for } i = 0, \ldots, m'-1. \tag{9.16}$$

The computation in Eq. (9.13) is described in vector form in Algorithm 7.

Algorithm 7 Structure of the new 4-way vectorisation of Poly1305 computation. Refer to Eq. (9.14, 9.15 and 9.16) for the definition of the vector quantities.

Input: $(D_0, \ldots, D_{\ell-1})$
Output: $\mathsf{Poly1305}_R(D_0, \ldots, D_{\ell-1})$

1: $\mathbf{T} \leftarrow \mathbf{D}_0$;
2: **for** $i = 1$ to $m' - 1$ **do**
3: $\quad \mathbf{T} \leftarrow \mathbf{R}_4 \circ \mathbf{T} + \mathbf{D}_i$
4: **end for**
5: $\mathbf{T} \leftarrow \mathbf{R} \circ \mathbf{T}$
6: **return** $T_0 + T_1 + T_2 + T_3$

In Algorithm 7 the entire computation can be performed using 4-way SIMD operations. In other words, by prepending 0's, the structure of the computation becomes balanced. It is possible to execute all the multiplications arising in Step 3 using $\mathsf{vecMult}(\cdot, \cdot)$ followed by $\mathsf{vecReduce}(\cdot)$.

Table 9.6: A snapshot of alignment of input for $\rho = 1$ if Goll-Gueron algorithm is followed

$\mathtt{t_0}$	$m_{0,2}$	$m_{0,0}$	0	0	0	0	0	0
$\mathtt{t_1}$	$m_{0,3}$	$m_{0,1}$	0	0	0	0	0	0
$\mathtt{t_2}$	x	$m_{0,4}$	x	0	x	0	x	0

For the case $\rho = 0$, the Step 8 of Algorithm 6 is not executed. In this case, Algorithms 6 and 7 become the same. For $\rho = 3$, there is a performance improvement of Algorithm 7 over Algorithm 6. For the cases of $\rho = 1$ and $\rho = 2$, the situation is more subtle. Directly using vecMult for the first multiplication in Algorithm 7 does not necessarily lead to speed gains. We address this issue in the next section.

Remark 9. *We have described Algorithm 7 for* 4*-decimated Horner computation. The same idea easily extends to* $\mathbf{d}$*-decimated Horner computation for any* $\mathbf{d} \geq 2$*.*

9.4.1 Improved Initial Multiplication

In Algorithm 7, the first multiplication is $\mathbf{R}_4 \circ \mathbf{D}_0$. Consider $\mathbf{D}_0 = (D_0, D_1, D_2, D_3)^\top$. Depending on the value of ρ, there are four cases.

$$\mathbf{D}_0 = \begin{cases} (M_0, M_1, M_2, M_3)^\intercal & \text{if } \rho = 0; \\ (0, 0, 0, M_0)^\intercal & \text{if } \rho = 1; \\ (0, 0, M_0, M_1)^\intercal & \text{if } \rho = 2; \\ (0, M_0, M_1, M_2)^\intercal & \text{if } \rho = 3. \end{cases} \tag{9.17}$$

Suppose $\rho = 1$ so that $\mathbf{D}_0 = (0, 0, 0, M_0)$. Let the 5-limb representation of M_0 be given by $(m_{0,4}, \ldots, m_{0,0})$. Consider the schematic of the operation vecMult as discussed in Section 9.3.1. The representation of $\mathbf{D}_0$ in the three 256-bit words $\mathtt{t_0}, \mathtt{t_1}$ and $\mathtt{t_2}$ will look as in Table 9.6 (where x is a don't care value). Since a lot of entries in the above representation are zeros, applying the Goll-Gueron vecMult operation to $\mathbf{R}_4$ and this $\mathbf{D}_0$ will result in the execution of a number of 32-bit multiplication operations whose results are known to be zero. By using a different representation for $\mathbf{D}_0$ and a different multiplication algorithm, it is possible to obtain the desired output using a substantially lower number of 32-bit SIMD multiplication operations. This leads to speed improvement.

A similar analysis shows that it is possible to obtain speed improvement also for the case $\rho = 2$ for which $\mathbf{D}_0 = (0, 0, M_0, M_1)$. When $\rho = 3$, $\mathbf{D}_0 = (0, M_0, M_1, M_2)$, and in this case, the number of zeros is not sufficient to provide any improvement by using a multiplication algorithm different from vecMult. Below we provide the top level schematics of the improved initial multiplication algorithms for the cases of $\rho = 1$ and $\rho = 2$.

Representation of $\mathbf{D}_0$ for the case $\rho = 1$ In this case, $\mathbf{D}_0 = (0, 0, 0, M_0)$ is represented using two 256-bit words as in Table 9.7. The five 256-bit words holding the output of $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{D}_0)$ in this case will have the form as in Table 9.8.

Table 9.7: A snapshot of alignment of input for $\rho = 1$ if the new algorithm is followed

$\mathtt{t_0}$	0	$m_{0,3}$	0	$m_{0,2}$	0	$m_{0,1}$	0	$m_{0,0}$
$\mathtt{t_1}$	0	$m_{0,4}$	0	0	0	0	0	0

Table 9.8: A snapshot of alignment of product for $\rho = 1$

S_0	$s_{0,3}$	0	0	0
S_1	$s_{1,3}$	0	0	0
S_2	$s_{2,3}$	0	0	0
S_3	$s_{3,3}$	0	0	0
S_4	$s_{4,3}$	0	0	0

Table 9.9: A snapshot of alignment of input for $\rho = 2$ if the new algorithm is followed

$\mathtt{t_0}$	$m_{0,3}$	$m_{1,3}$	$m_{0,2}$	$m_{1,2}$	$m_{0,1}$	$m_{1,1}$	$m_{0,0}$	$m_{1,0}$
$\mathtt{t_1}$	0	$m_{1,4}$	0	$m_{0,4}$	0	0	0	0

Table 9.10: A snapshot of alignment of product for $\rho = 2$

S_0	$s_{0,3}$	$s_{0,2}$	0	0
S_1	$s_{1,3}$	$s_{1,2}$	0	0
S_2	$s_{2,3}$	$s_{2,2}$	0	0
S_3	$s_{3,3}$	$s_{3,2}$	0	0
S_4	$s_{4,3}$	$s_{4,2}$	0	0

Representation of $\mathbf{D}_0$ for the case $\rho = 2$ In this case, $\mathbf{D}_0 = (0, 0, M_0, M_1)$ is represented using two 256-bit words as in Table 9.9.

The five 256-bit words holding the output of $\mathsf{vecMult}(\mathbf{R}_4, \mathbf{D}_0)$ in this case will have the form as in Table 9.10. In both the cases, the representation of $\mathbf{R}_4$ using two 256-bit words $\mathtt{r_0}$ and $\mathtt{r_1}$ remain the same as in Section 9.3.1. The multiplication algorithms for the above two cases apply SIMD permutations, 32-bit SIMD multiplications and 64-bit SIMD additions to produce the required output $\mathbf{S}$ in five 256-bit words as shown above. The vecReduce algorithm mentioned in Section 9.3.1 is applied to $\mathbf{S}$ to obtain the result $\mathbf{R}_4 \circ \mathbf{D}_0$. The output of vecReduce is in the form of three 256-bit words which is stored in $\mathbf{T}$. The further multiplications $\mathbf{R}_4 \circ \mathbf{T}$ in the loop at Step 2 of Algorithm 7 are performed using the algorithm vecMult and vecReduce.

Remark 10. *For the case $\rho = 2$, there are several variants of the initial multiplication algorithm which avoid multiplications by zero. For all such variants the number of _mm256_mul_epu32 is 13.*

9.4.2 Lazy Reduction

The lazy reduction strategy described in Section 9.3.2 applies to the loop in Steps 2 - 4 of Algorithm 7. The computation is divided into groups where each group processes λ of the $\mathbf{D}_i$'s. As in the case of Algorithm 6, the lazy reduction strategy requires only a $1/\lambda$ fraction of the number of reductions required in a direct implementation of Algorithm 7. Following the code for [60], we have incorporated the lazy reduction strategy for messages having at least 832 bytes with values of λ to be 2 or 3.

9.5 Implementation and Comparison

We have implemented the SIMD strategy given in Algorithm 7 for evaluation of the Poly1305 hash function. This implementation consisted of modifying the Intel intrinsics code implementing the SIMD strategy in [60]. Portions of the code were used without any change. In particular, the basic 4-way multiplication routine of [60] has been directly used. On the other hand, the improved initial multiplication algorithms are new to our SIMD strategy and had to be implemented. The modified code is publicly available at the following link.

`https://github.com/Sreyosi/Improved-SIMD-Implementation-of-Poly1305`

Performance has been measured in terms of number of machine cycles per byte under the same conditions as mentioned in [60]: Intel Turbo Boost Technology, Intel Hyper-Threading Technology and Intel Speedstep Technology were disabled. Performances of the new code and that of the code accompanying [60] were measured using the same strategy.

Measurements were made on the following platforms:

- Haswell: Intel Core i7-4790 CPU @ 3.60GHz x 8; running Ubuntu 18.04.2 LTS (64-bit); gcc version 7.4.0; Clang version 8.4.
- Skylake: Intel Core i7-6500U CPU @ 2.50GHz x 2; running Ubuntu 14.04 LTS (64-bit); gcc version 5.5.0; Clang version 8.4.
- Kaby Lake: Intel Core i7-7700U CPU @ 3.60GHz x 4; running Ubuntu 18.04 LTS (64-bit); gcc version 7.3.0; Clang version 8.4.

In all cases, measurements were made on a single core of the specified machines. The compile command used was the following:

- `gcc -mavx2 -O3 -fomit-frame-pointer`
- `gcc -mavx2 -O2 -fomit-frame-pointer`

Table 9.11: A summary of the comparative performance analysis of the new code with the code of [60] in gcc compiler

Processor	Range	Max speed-up	Avg speed-up
Haswell	49B - 1000B	29.70%	12.06%
	1001B - 2000B	22.31%	12.46%
	2001B - 4000B	15.15%	9.06%
Skylake	49B - 1000B	36.81%	15.05%
	1001B - 2000B	15.24%	6.94%
	2001B - 4000B	12.66%	3.29%
Kaby Lake	49B - 1000B	35.33%	13.12%
	1001B - 2000B	21.49%	12.94%
	2001B - 4000B	21.17%	10.51%

Message length For measuring performance and comparison to [60], we considered messages with lengths up to 4KB[4]. If the number of 16-byte blocks in the padded message is a multiple of 4, then the new code becomes exactly the Goll-Gueron code. Consequently, there is no difference of performance for such message lengths.

In view of the above, for the purpose of comparing the performance of the new code to the Goll-Gueron code, we considered message lengths from 49 bytes to 4000 bytes which are not divisible by 64. For each message length, we have taken measurements of both the Goll-Gueron code and the new code. Suppose that for a message length l bytes, the Goll-Gueron code requires t_0 cycles/byte and the new code requires t_1 cycles/byte. Then the speed-up (in percentage) attained for message length l is $\mathsf{su}_l = 100(t_1 - t_0)/t_0$. The average speed-up is the average of all the su_l's.

A top-level summary of the comparison is as follows.

- Haswell: speed-up has been obtained in 99.87% cases of message lengths that were considered; in 0.07% cases, the performances of both the codes were the same; in 0.05% cases, the new code has shown a slight slowdown.
- Skylake: speed-up has been obtained in 89.51% cases of the message lengths that were considered; in 6.27% cases, the performances of both the codes were the same; in 4.21% cases, the new code has shown a slight slowdown.
- Kaby Lake: speed-up has been obtained in 99.66% cases of the message lengths that were considered; in 0.17% cases, the performances of both the codes were the same; in 0.15% cases, the new code has shown a slight slowdown.

Table 9.11 provides a summary of the maximum and average speed-ups for the three platforms for three different ranges of message lengths.

Comparisons in terms of abolute input lengths are given in Table 9.12 and Table 9.13.

[4]See `http://www.caida.org/data/passive/trace_stats/nyc-A/2019/equinix-nyc.dirA.20190117-130000.UTC.df.xml` of the Center for Applied Internet Data Analysis for the relevance of short messages in IPv4 and IPv6 traffic. Accessed on 27th June, 2019.

Table 9.12: A summary of the comparative performance analysis of the new code with the code of [60] in Clang compiler

Processor	Range	Max speed-up	Avg speed-up
Haswell	49B - 1000B	23.33%	12.92%
	1001B - 2000B	14.52%	5.27%
	2001B - 4000B	10.20%	2.44%
Skylake	49B - 1000B	29.61%	12.95%
	1001B - 2000B	21.49%	8.99%
	2001B - 4000B	20.45%	5.60%
Kaby Lake	49B - 1000B	29.43%	10.57%
	1001B - 2000B	18.58%	4.54%
	2001B - 4000B	10.84%	1.64%

Table 9.13: Performance observed in Haswell microarchitecture using Clang and gcc compilers seperately

	Clang			GCC		
length	Goll-Gueron	New	speed-up	Goll-Gueron	New	speed-up
56	4.71	4.36	7.43 %	4.71	4.36	7.43%
80	3.85	4.00	3.75%	3.70	3.50	5.40%
96	3.33	3.25	2.40%	3.17	2.96	6.62%
112	3.02	2.61	13.57%	3.11	2.64	15.11%
160	3.32	2.96	10.84%	2.30	2.17	5.62%
224	2.00	1.88	6.00%	1.88	1.79	4.78%
240	2.08	1.78	14.42%	1.93	1.70	13.52%
496	1.43	1.29	9.79%	1.37	1.24	9.48%
600	1.37	1.21	11.67%	1.31	1.21	7.63%
800	1.14	1.00	12.28%	1.11	1.09	1.80%
1000	1.26	1.07	15.07%	1.28	1.17	8.59%
2000	0.92	0.92	0%	0.94	0.89	5.31%

9.6 Details of Performance Comparison

For Haswell, Figure 9.1 shows the plot of the speed-up of the new code over the Goll-Gueron code as the message length varies; the actual values of the cycles/byte measure are shown in Figures 9.2 to 9.4. For Skylake, Figure 9.5 shows the plot of the speed-up of the new code over the Goll-Gueron code as the message length varies; the actual values of the cycles/byte measure are shown in Figures 9.6 to 9.8. For Kaby Lake, Figure 9.9 shows the plot of the speed-up of the new code over the Goll-Gueron code as the message length varies; the actual values of the cycles/byte measure are shown in Figures 9.10 to 9.12.

Table 9.14: Performance observed in Skylake microarchitecture using Clang and gcc compilers seperately

	Clang			GCC		
length	Goll-Gueron	New	speed-up	Goll-Gueron	New	speed-up
56	4.57	4.11	10.06%	4.39	3.79	13.66%
80	3.48	3.40	2.29%	4.93	4.64	5.88%
96	2.98	2.92	2.01%	2.77	2.75	0.72%
112	3.02	2.61	13.57%	2.80	2.43	13.21%
160	2.17	2.09	3.68%	2.05	1.94	5.36%
224	2.92	2.08	28.76%	1.66	1.61	3.01%
240	1.90	1.62	14.73%	1.76	1.56	11.36%
496	1.26	1.14	9.52%	1.21	1.14	5.78%
600	1.20	1.07	10.83%	1.19	1.08	9.24%
800	0.99	0.95	4.04%	0.97	0.98	-1.03%
1000	1.03	0.93	9.70%	1.01	0.92	8.91%
2000	0.78	0.77	1.28%	0.76	0.78	-2.63%

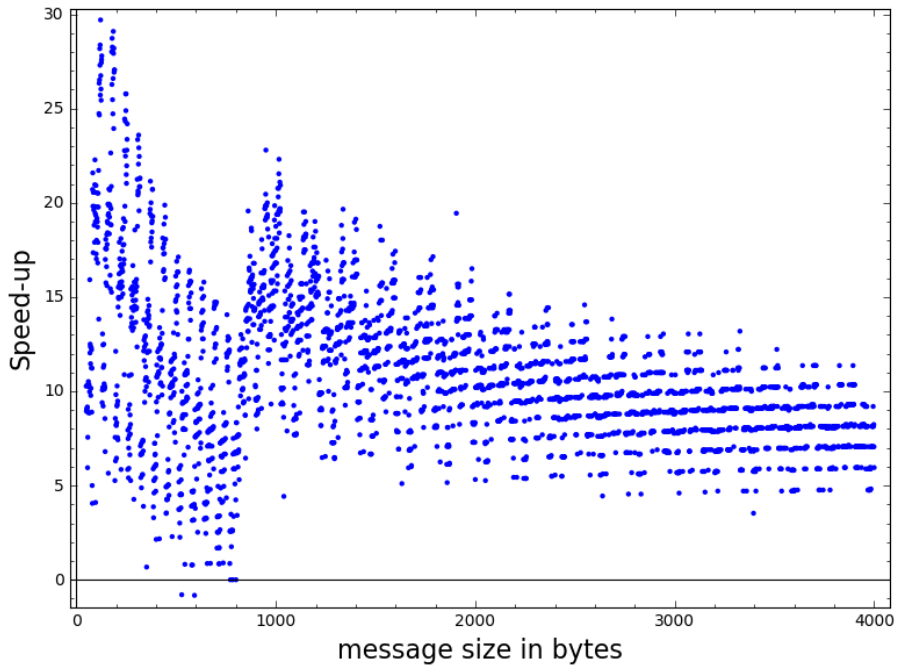


Figure 9.1: Speed-up vs message size in bytes for Haswell.

Table 9.15: Performance observed in Kaby Lake microarchitecture using Clang and gcc compilers seperately

	Clang			GCC		
length	Goll-Gueron	New	speed-up	Goll-Gueron	New	speed-up
56	4.57	4.11	10.06%	4.43	3.79	14.44%
80	3.55	3.50	1.40%	3.27	3.15	3.66%
96	2.94	2.90	1.36%	2.94	2.90	1.36%
112	2.98	2.64	11.04%	2.86	2.27	20.62%
160	2.17	2.09	3.68%	2.05	1.90	7.31%
224	1.78	1.67	6.17%	1.64	1.54	6.09%
240	1.85	1.61	12.97%	1.75	1.44	17.71%
496	1.21	1.12	7.43%	1.20	1.21	-0.83%
600	1.17	1.08	7.69%	1.16	1.05	9.48%
800	0.94	0.97	-3.19%	0.94	0.97	-3.19%
1000	1.04	0.93	11.00%	1.02	0.93	8.82%
2000	0.78	0.77	1.28%	0.77	0.77	0%

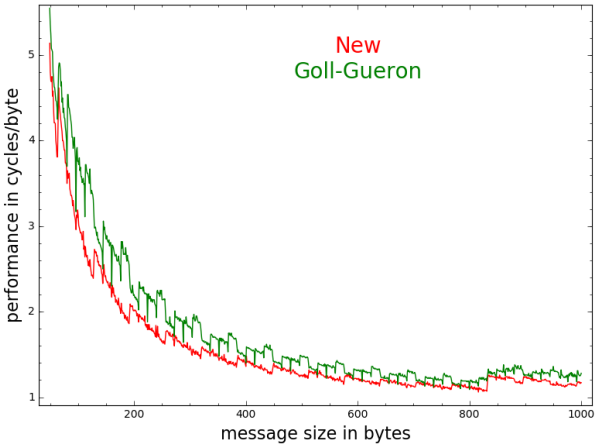


Figure 9.2: cycles/byte vs message size in bytes (49 - 1000 bytes) for Haswell.

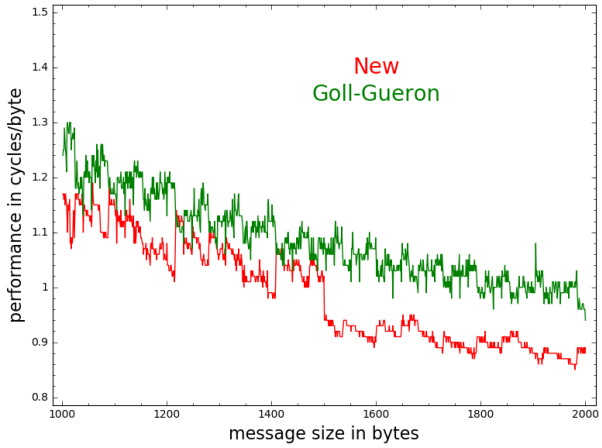


Figure 9.3: cycles/byte vs message size in bytes (1001 - 2000 bytes) for Haswell.

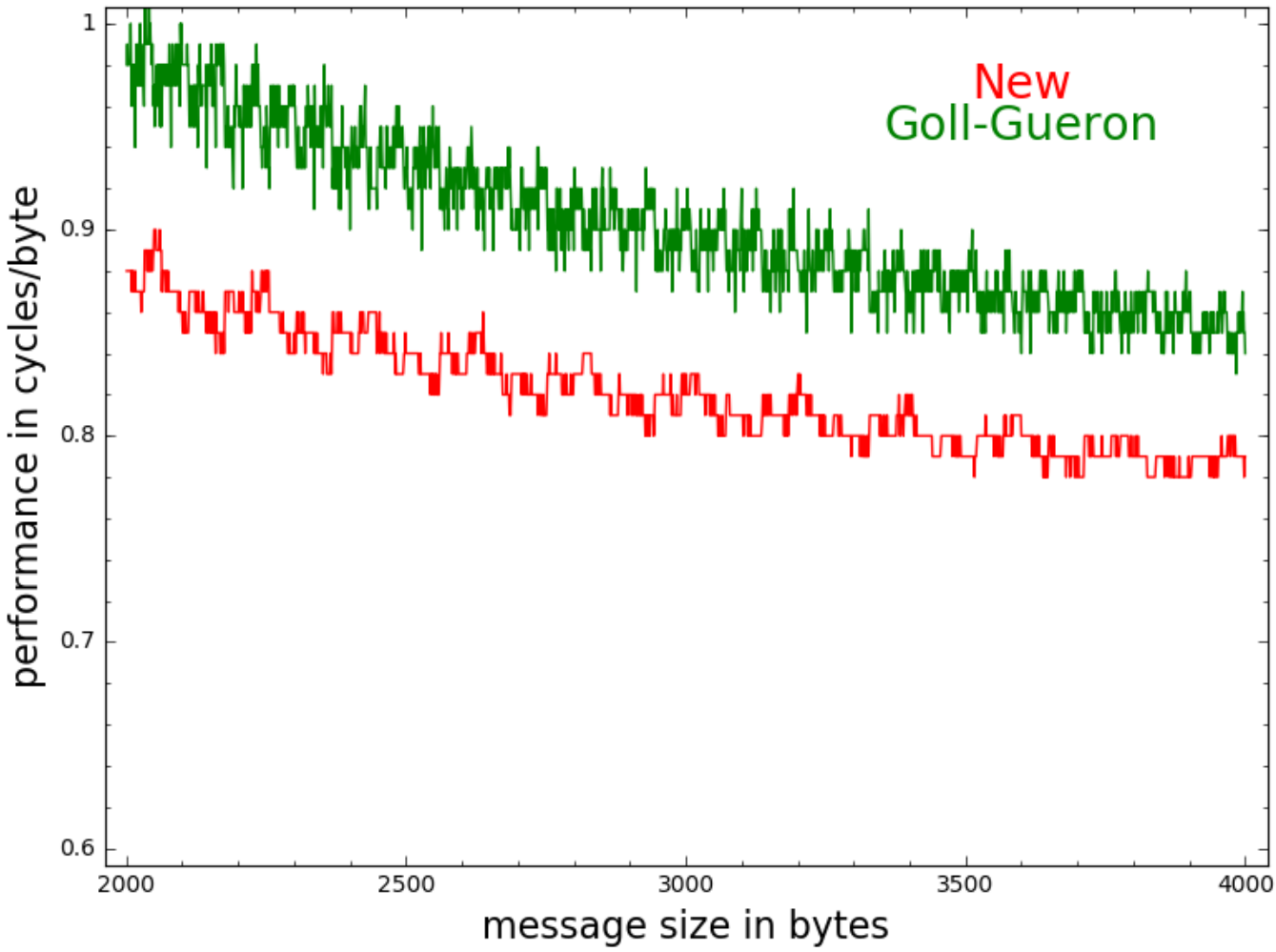


Figure 9.4: cycles/byte vs message size in bytes (2001 - 4000 bytes) graph for Haswell.

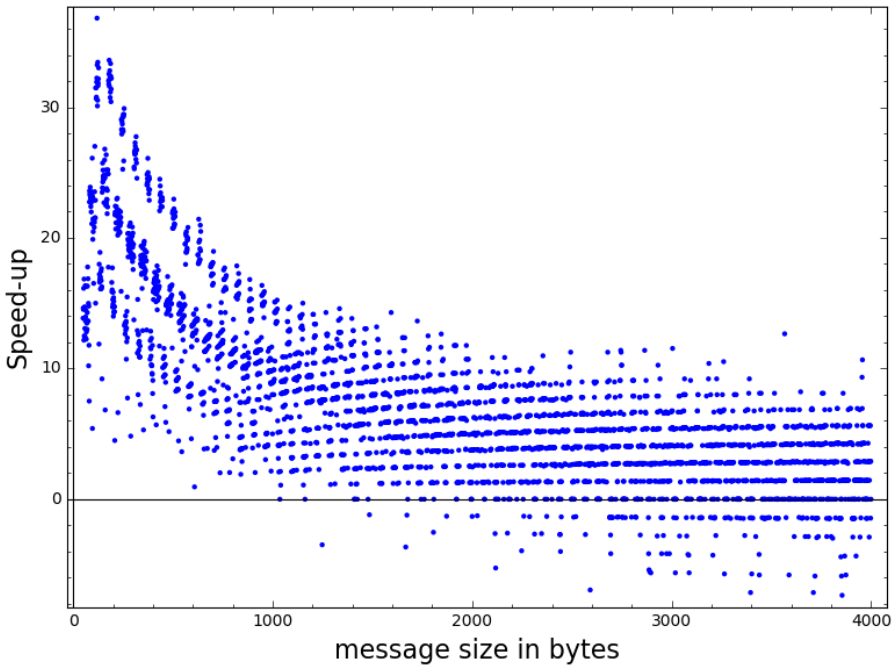


Figure 9.5: Speed-up vs message size in bytes graph for Skylake.

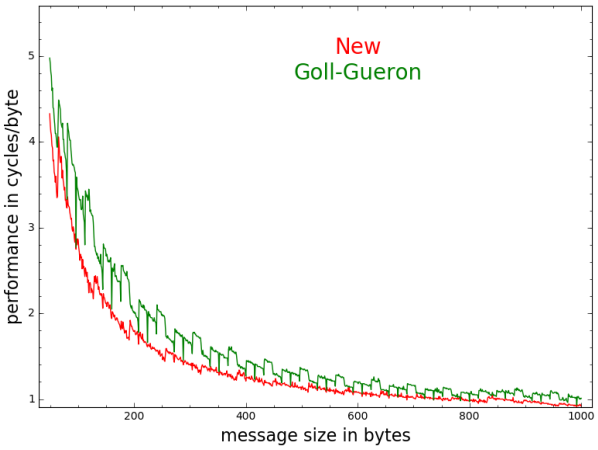


Figure 9.6: cycles/byte vs message size in bytes (49 - 1000 bytes) graph for Skylake.

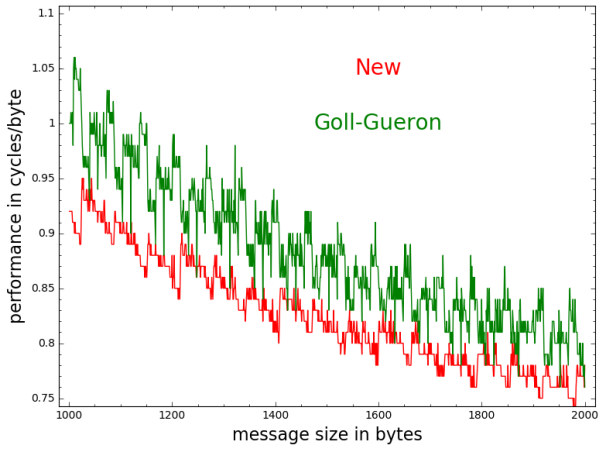


Figure 9.7: cycles/byte vs message size in bytes (1001 - 2000 bytes) graph for Skylake.

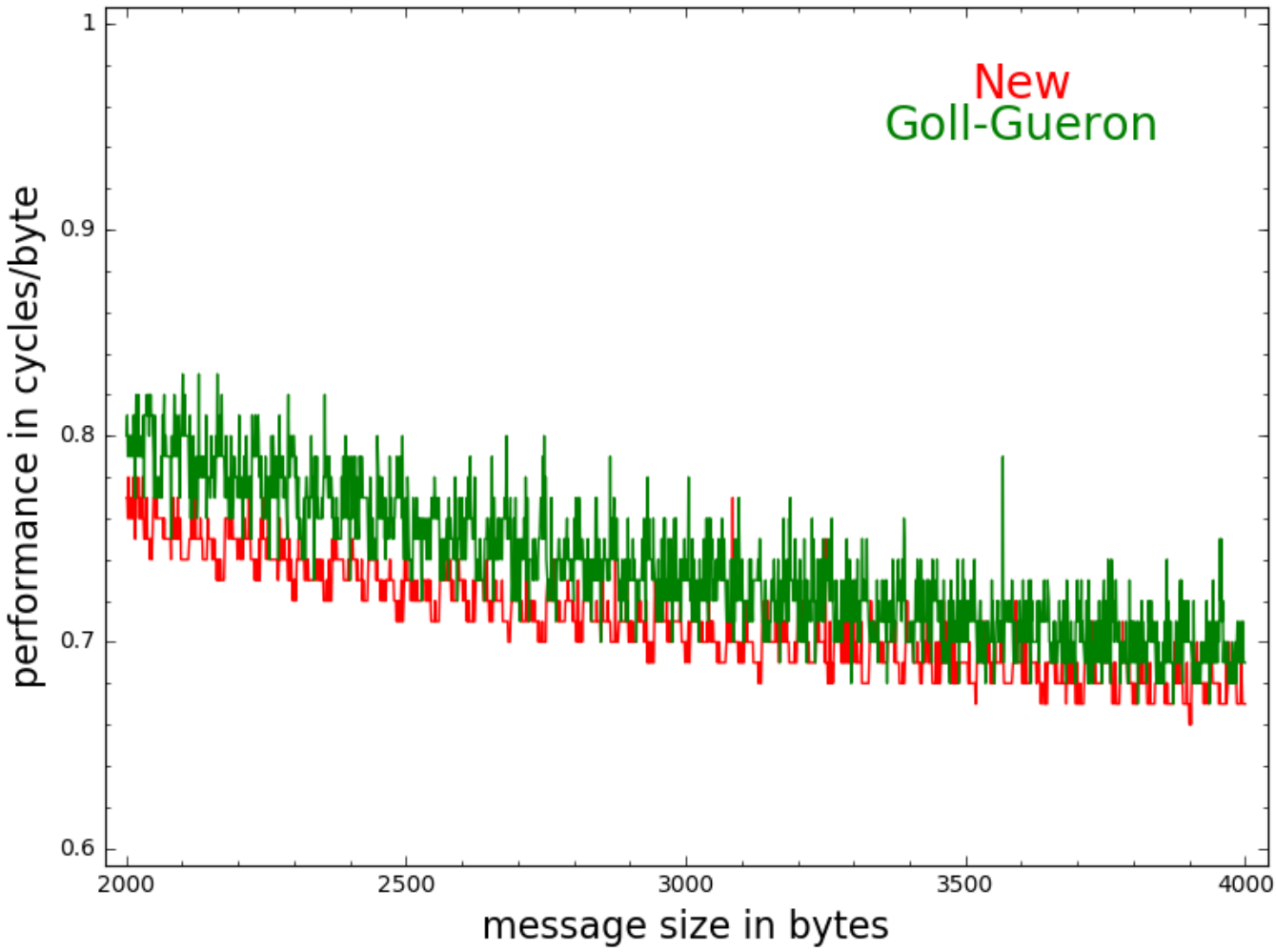


Figure 9.8: cycles/byte vs message size in bytes (2001 - 4000 bytes) graph for Skylake.

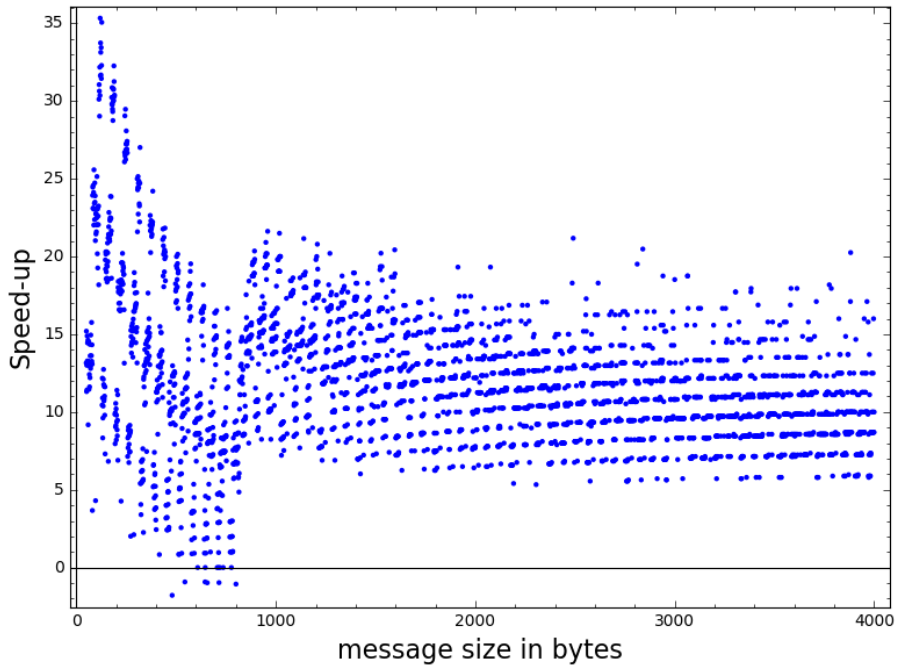


Figure 9.9: Speed-up vs message size in bytes graph for Kaby Lake.

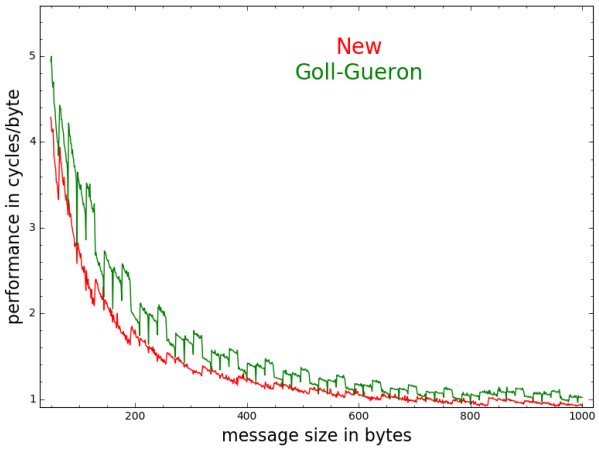


Figure 9.10: cycles/byte vs message size in bytes (49 - 1000 bytes) graph for Kaby Lake.

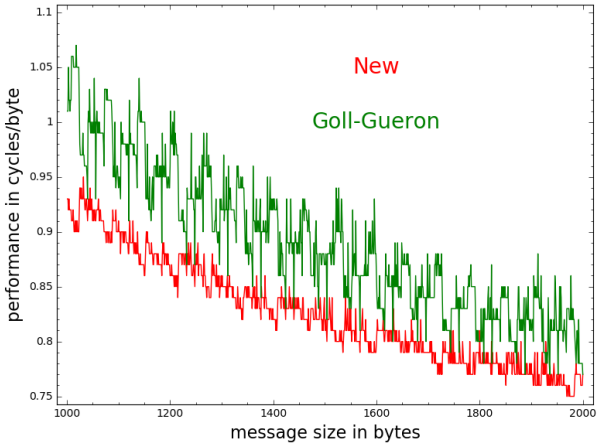


Figure 9.11: cycles/byte vs message size in bytes (1001 - 2000 bytes) graph for Kaby Lake.

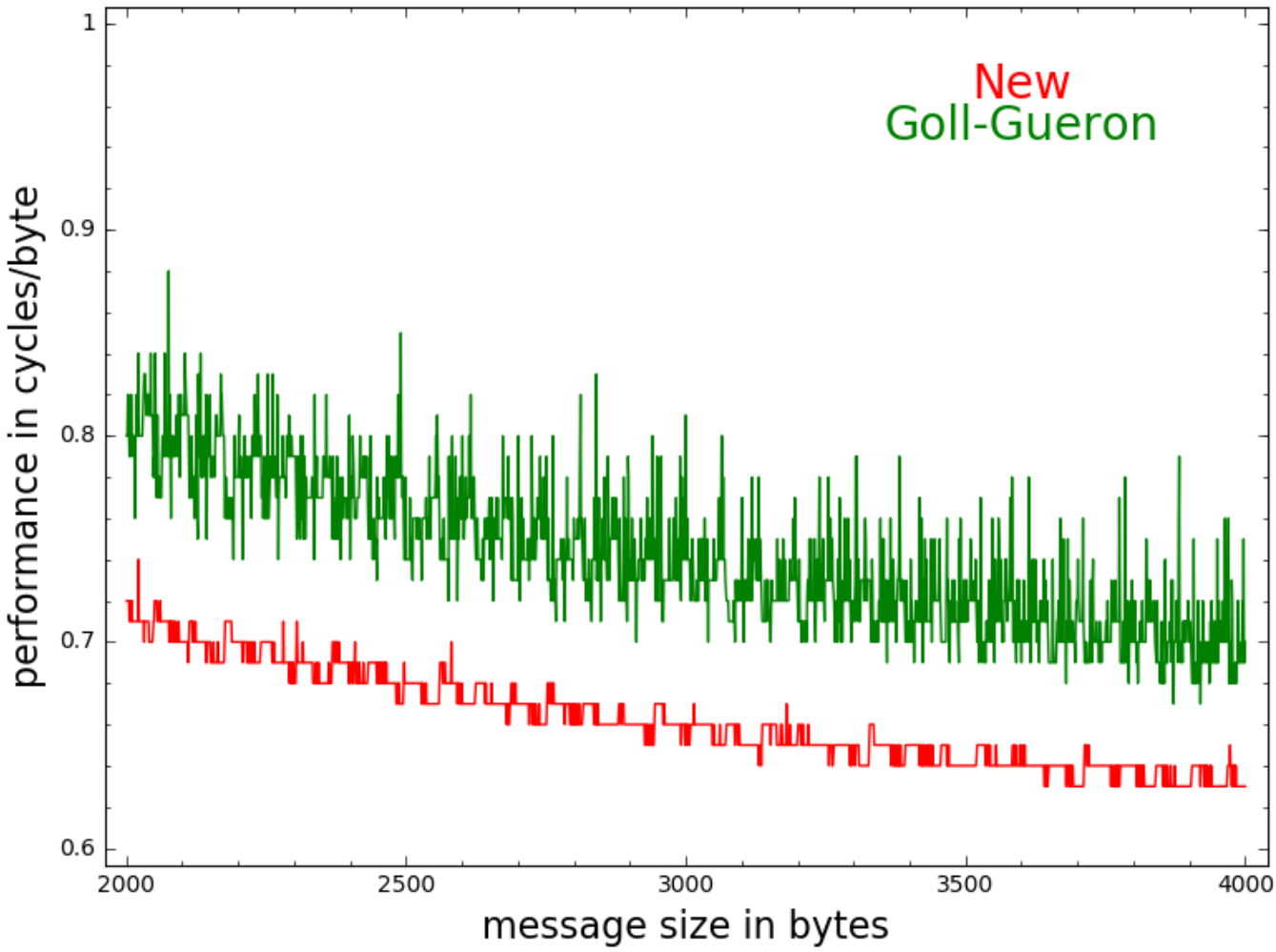


Figure 9.12: cycles/byte vs message size in bytes (2001 - 4000 bytes) graph for Kaby Lake.

9.7 Summary

In this work, we have proposed a simple modification to the previous Goll-Gueron strategy for SIMD implementations of the Poly1305 algorithm. Implementation of the modified algorithm shows noticeable speed improvements on modern Intel processors for short messages when the number of blocks is not a multiple of 4 using AVX2 intrinsics.

Chapter 10

Polynomial Hashing Over Prime Order Fields

In Chapter 1 and Chapter 8 we have mentioned that universal hash functions are important primitives for designing certain modes of operation like message authentication codes and authenticated encryption schemes. The importance of univariate polynomials for building such hash functions has been discussed in Chapter 8. One of the factors that heavily influence the performance of polynomial-evaluation based hash functions, in terms of *cycles per byte*, is the efficiency of the evaluation of the polynomial at a point. To improve the efficiency of such evaluations SIMD computations are considered. In Chapter 9 we have proposed an improved SIMD implementation of Poly1305. Apart from aiming to speed up the multiplication routines, reducing the number of field multiplications can bring down the *cycles per byte* of a hash function. In this chapter we consider various trade-offs which give us new and better hash functions than Poly1305. We consider trade-offs between two evaluation techniques and trade-offs for choice of prime order fields.

Two interesting techniques of evaluating such polynomials are- Horner's rule and evaluation technique of BRW polynomials. To distinguish between the two methods of constructing the polynomials, we will denote the polynomials obtained by the method of [42, 103, 104] by Poly (and call these the usual polynomials) and denote the polynomials obtained by the method of [17] by BRW. In this chapter we make a comprehensive study of polynomial hashing based on both of these methods. Poly and BRW and their various combinations over a prime order field $\mathbb{F}_p$.

The type of univariate polynomial used in the construction of Poly1305 can be evaluated efficiently by Horner's method. On the other hand BRW polynomials [14] based on [92] can be evaluated faster than Poly1305. Theoretically, for the same input string X the evaluation of BRW polynomial requires about half the number of field multiplications compared to the evaluation of the usual polynomial Poly. To the best of our knowledge, till date there has been no concrete proposal of BRW based hash functions over prime order fields. Concrete BRW based hash functions and their implementations in hardware and software have been proposed over binary extension fields [94, 33, 34, 32, 59]. Due to lesser number of field multiplications required it is interesting to study the performance of a hash function based on BRW polynomial defined over the prime order field $\mathbb{F}_{2^{130}-5}$ and to compare the performance

in terms of cycles per byte with that of Poly1305.

Apart from the prime $2^{130} - 5$, another prime number $2^{127} - 1$ has been considered for BRW polynomial evaluation over prime order fields. For mod $2^{127} - 1$ arithmetic each element of $\mathbb{Z}_{2^{127}-1}$ requires two 64-bit registers if saturated limb representation (please refer to Section 7.3) is followed whereas the former requires three 64-bit registers for representing any element of $\mathbb{Z}_{2^{130}-5}$ in saturated limb representation. One of the major factors which decide the cost of any field operation is the number of limbs required for representing a field element. So studying the performance of BRW-based hash function defined over $\mathbb{Z}_{2^{127}-1}$ is interesting.

Concrete instantiations of the hash functions are proposed for the primes $2^{127} - 1$ and $2^{130} - 5$. The Poly based hash function for the prime $2^{130} - 5$ coincides with Poly1305 for messages whose lengths are multiples of eight. The Poly based hash for the prime $2^{127} - 1$ is new (see below for a discussion on previous proposals of usual polynomial based hashing for this prime). As far as we know, none of the other hash functions proposed in this work have been reported in the literature. The prime $2^{127} - 1$ was suggested by Taylor [103] for instantiating Poly based hashing. Concrete instantiations were proposed by Bernstein [13] and Kohno, Viega and Whiting [70]. The construction in [13] used 32-bit coefficients and floating point arithmetic for implementation, while the construction in [70] used 96-bit coefficients and integer arithmetic for implementation. In contrast, we use 126-bit coefficients (obtained by padding 120-bit message blocks) and integer arithmetic for implementation. To speed-up evaluation, [13] used large precomputed tables ("few kilobytes of data for each key", as mentioned in [16]). In contrast, in our implementation, for speeding up the computation of Poly using grouped Horner with group size 8, only 128 bytes of pre-computed data are required for each key. Apart from [103, 13, 70], we know of no other previous work which considered the prime $2^{127} - 1$ for polynomial hashing. In the Poly1305 paper [16], under 'Design decisions', Bernstein writes "I considered various primes above 2^{128}". No discussion is provided on the reason for discarding the prime $2^{127} - 1$ used by Bernstein in his previous work [13]. Our proposal of Poly based hash function using $2^{127} - 1$ and its implementation suggests that $2^{127} - 1$ is a much faster option compared to $2^{130} - 5$.

A major aspect of our work is implementation. Horner's rule is the usual method of evaluating Poly. A previous suggestion by Gueron [61] in the context of binary extension fields had put forward a method, which we call grouped Horner, to eliminate some operations from Horner's rule by pre-computing certain key powers and performing the evaluation of Poly in groups of coefficients. We implemented both basic and grouped Horner's rule for both the primes. The definition of BRW is recursive which makes it difficult to implement efficiently. A non-recursive algorithm to evaluate BRW was proposed in [59] and an implementation over binary extension fields was reported. We simplify the algorithm from [59]. The simplification makes substantial changes to the manner in which intermediate quantities are stored and accessed. As a result, the proof of correctness provided in [59] no longer applies to the modified algorithm. So we provide a detailed proof of correctness of the new algorithm. We have implemented this new algorithm for both the primes.

We define four hash functions over $\mathbb{F}_p$, one based only on Poly, a second one based only on BRW and two others which use a combination of BRW and Poly. We have made 64-bit assembly implementation of all the hash functions for both the primes for all prac-

tical values of design and implementation parameters. The code is targeted towards the Intel Broadwell and later generation processors. In particular, along with the usual integer arithmetic instructions, we use the instructions `mulx`, `adcx` and `adox` which are available from the Broadwell processor onwards. To the best of our knowledge, there is no previous implementation of Poly1305 using these instructions.

The extensive implementation exercise provides some important insights.

1. The hash functions defined using $2^{127} - 1$ are faster than the corresponding hash functions defined using $2^{130} - 5$. In particular, the Poly based hash function over $2^{127} - 1$ is faster than Poly1305 and the speed improvements range from 10% to 30%.

2. While in theory, BRW evaluation requires about half the number of field multiplications compared to Poly evaluation, this is not true in practice. Compared to Horner's rule based evaluation of Poly, BRW produces a speed improvement of 30% to 40% for messages which are not too short. If, however, grouped Horner is used for evaluating Poly, then the speed improvement becomes around 10%. While this is much less than what is predicted by theory, a speed improvement of 10% is indeed significant in practice.

Our final hash function proposal is a combination of Poly and BRW over the prime $2^{127} - 1$. If the number of (appropriate sized) blocks in the message is less than 16, then Poly is used. If the number of blocks is at least 16, then a 2-level hash function is used, where BRW is used at the first level and Poly is used at the second level. Timing data from our implementations show that the new hash function is significantly faster than Poly1305, achieving speed improvements of about 8.5% (for 10-byte messages) to about 40% (for 5000-byte messages).

Theoretical properties of these hash functions including explicit upper bounds on their differential probabilities are derived.

From timing results, we observe that for short messages, the Poly based hash function has the best performance, while for longer messages it becomes slower than the other hash functions. To obtain the best performance for messages of all lengths, we put forward a combination hash function which applies the Poly based hash function for short messages and switches to another hash function for longer messages. We show that the combination hash function is secure by obtaining an upper bound on the corresponding differential probability.

This chapter is based on the work in [22].

10.1 Preliminaries

Let X be a binary string. A polynomial hashing strategy constructs a univariate polynomial over a finite field from X and evaluates the polynomial at a secret point of the field, called the key, to obtain the digest. The security requirement on such a hash function is that all differential probabilities are provably small, where a differential probability is the probability,

Table 10.1: The parameters m, k and n for the two values of p considered in this work.

p	m	k	n
$2^{127}-1$	127	126	120
$2^{130}-5$	130	128	128

Table 10.2: The parameters m, k and n for the two values of p considered in this work.

over a random choice of the key, that the difference of the digests corresponding to two different strings is equal to an arbitrary element of the field.

Let $\mathbb{F}$ be a finite field. For a non-zero polynomial $P(x) \in \mathbb{F}[x]$, by $\mathsf{deg}(P(x))$ we will denote the degree of $P(x)$. For $i \geq 0$ and $M_1, \ldots, M_i \in \mathbb{F}$, we define polynomials $\mathsf{Poly}(x; M_1, \ldots, M_i)$ and $\mathsf{BRW}(x; M_1, M_2, \ldots, M_i)$ in $\mathbb{F}[x]$ with indeterminate x and parameters $M_1, \ldots, M_i$. Please note that the definition of Horner in Chapter 7 is same as that in Eq. 10.1. It has been restated in this chapter for the sake of completeness.

$$\mathsf{Poly}(x; M_1, \ldots, M_i) \;=\; \begin{cases} 0, & \text{if } i = 0; \\ M_1 x^{i-1} + M_2 x^{i-2} + \cdots + M_{i-1}x + M_i, & \text{if } i > 0, \end{cases} \tag{10.1}$$

The definition of $\mathsf{BRW}(x; M_1, M_2, \ldots, M_i)$ is same as that in Chapter 7 but here the evaluation technique is non-recursive.

For $i \geq 3$, $\mathsf{BRW}(x; M_1, M_2, \ldots, M_i)$ is a monic polynomial. The following has been proved in [17].

Theorem 1. *[17]*

1. *For every $i \geq 0$, the map from $\mathbb{F}^i$ to $\mathbb{F}[x]$ given by $(M_1, \ldots, M_i) \mapsto \mathsf{BRW}(x; M_1, \ldots, M_i)$ is injective.*

2. *For $i \geq 1$, let $\mathfrak{d}(i)$ denote $\mathsf{deg}(\mathsf{BRW}(x; M_1, \ldots, M_i))$. For $i \geq 3$, $\mathfrak{d}(i) = 2^{\lfloor \lg i \rfloor + 1} - 1$ and so $\mathfrak{d}(i) \leq 2i - 1$; the bound is achieved if and only if $i = 2^a$; and $\mathfrak{d}(i) = i$ if and only if $i = 2^a - 1$ for some integer $a \geq 2$ where logarithm to base two is denoted by* lg.

3. *For $\tau \in \mathbb{F}$ and $i \geq 3$, $\mathsf{BRW}(\tau; M_1, \ldots, M_i)$ can be computed using $\lfloor i/2 \rfloor$ field multiplications and $\lfloor \lg i \rfloor$ additional field squarings to compute $\tau^2, \tau^4, \ldots$.*

10.2 Constructions

Let p be a prime and $\mathbb{F}_p$ be the finite field of order p. We will work with the primes $2^{127} - 1$ and $2^{130} - 5$. Given the prime p, we define the integers m, k and n as shown in Table 10.2. Elements of $\mathbb{F}_p$ can be represented as m-bit strings. Since k and n are less than m, we will consider k-bit and n-bit strings to represent elements of $\mathbb{F}_p$, where the most significant $m-k$ and $m-n$ bits respectively are set to 0. We will also adopt the usual convention that the binary representation of a non-negative integer is written with the least significant bit on

the right. For a positive integer i and $0 \leq j < 2^i$, by $\mathsf{bin}_i(j)$ we will denote the i-bit binary representation of j. For example, if $i = 4$ and $j = 13$, then $\mathsf{bin}_i(j)$ is the string 1101.

Formatting and padding: A binary string X of length $L \geq 0$ is formatted (or partitioned) into ℓ blocks $X_1, \ldots, X_\ell$, where the length of X_i is n for $1 \leq i \leq \ell - 1$, the length of X_ℓ is s with $1 \leq s \leq n$, and $X = X_1||X_2|| \cdots ||X_\ell$. Note that if X is the empty string, i.e. if $L = 0$, then $\ell = 0$. If the length of a block is n, then we call it a full block, otherwise we call it a partial block. By $\mathsf{format}(X)$ we will denote the list $(X_1, \ldots, X_\ell)$ obtained from X using the above described procedure. We describe two padding schemes.

- $\mathsf{pad1}(X_1, \ldots, X_\ell)$ returns $(M_1, \ldots, M_\ell)$, where $M_i = 0^{m-n-2}||1||X_i$, for $i = 1, \ldots, \ell - 1$, and $M_\ell = 0^{m-s-2}||1||X_\ell$.
- $\mathsf{pad2}(X_1, \ldots, X_\ell)$ returns $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$, where $M_i = 0^{m-n-1}||X_i$, for $i = 1, \ldots, \ell - 1$, $M_\ell = 0^{m-s-1}||X_\ell$.

Remark 11. *For both the padding schemes, the length of each M_i, $i = 1, \ldots, \ell$, is $m-1$ and we consider M_i to be an element of $\mathbb{F}_p$. Also, $\mathsf{bin}_{m-1}(L)$ is considered to be an element of $\mathbb{F}_p$. If $\ell = 0$, then the input list is empty;* $\mathsf{pad1}()$ *returns the empty list, while* $\mathsf{pad2}()$ *returns a singleton list containing 0^{m-1}.*

In Section 10.3.1, we provide explanations for the choice of the padding schemes.

We describe four constructions of hash function families, namely polyHash, BRWHash, t-BRWHash, and d-2LHash. The key space and digest space for all the four families are the following. The key space is $\{0,1\}^k$; τ denotes the k-bit key which is considered to be an element of $\mathbb{F}_p$. The digest space is the group $(\mathbb{Z}_{2^k}, +)$. The message space for polyHash is the set of all binary strings, while the message space for the other three hash function families is the set of all binary strings of lengths less than 2^{m-1}. See, however, Remark 15 in Section 10.3 for further discussion on the message length.

In the descriptions below, X denotes a message which is a binary string of length $L \geq 0$.

Construction polyHash**:** Given a binary string X, let $(M_1, \ldots, M_\ell)$ be the output of $\mathsf{pad1}(\mathsf{format}(X))$. We define

$$\mathsf{polyHash}_\tau(X) \;=\; (P_1(\tau; M_1, \ldots, M_\ell) \bmod p) \bmod 2^k, \tag{10.2}$$

where $P_1(x; M_1, \ldots, M_\ell)$ is a polynomial in $\mathbb{F}_p[x]$ defined as follows.

$$P_1(x; M_1, \ldots, M_\ell) \;=\; x \cdot \mathsf{Poly}(x; M_1, \ldots, M_\ell). \tag{10.3}$$

The family polyHash is motivated by the Poly1305 [16] construction and for the prime $2^{130}-5$, if X is a sequence of bytes, then polyHash is exactly the same as Poly1305.

Construction BRWHash**:** Given a binary string X, let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$. We define

$$\mathsf{BRWHash}_\tau(X) \;=\; (P_2(\tau; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \bmod p) \bmod 2^k, \tag{10.4}$$

where $P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is a polynomial in $\mathbb{F}_p[x]$ defined as follows.

$$P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \;=\; x(x \cdot \mathsf{BRW}(x; M_1, \ldots, M_\ell) + \mathsf{bin}_{m-1}(L)). \quad (10.5)$$

Construction t-BRWHash**:** This construction is parameterised by an integer $t \geq 2$. Let $\mathfrak{m} = \ell - (\ell \bmod 2^t)$. Given a binary string X, let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$. We define

$$t\text{-}\mathsf{BRWHash}_\tau(X) \;=\; (P_3(\tau; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \bmod p) \bmod 2^k, \quad (10.6)$$

where $P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is a polynomial in $\mathbb{F}_p[x]$ defined as follows.

$$\begin{aligned} &P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \\ &\quad = \; x \cdot \mathsf{Poly}(x; \mathsf{BRW}(x; M_1, M_2, \ldots, M_{\mathfrak{m}}), M_{\mathfrak{m}+1}, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)). \end{aligned} \quad (10.7)$$

Note that $\mathfrak{m}$ is a multiple of 2^t. The idea in t-BRWHash is to process the first $\mathfrak{m}$ blocks using BRW to obtain a single output and then combine this output with the leftover $\ell - \mathfrak{m}$ blocks and the length block using Poly.

Construction d-2LHash**:** This construction is parameterised by an integer $d \geq 2$. Let $\delta = 2^d - 1$ and $\mathfrak{n} = \lfloor \ell/\delta \rfloor$. Given a binary string X, let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$. We define

$$d\text{-}\mathsf{2LHash}_\tau(X) \;=\; (P_4(\tau; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \bmod p) \bmod 2^k, \quad (10.8)$$

where the polynomial $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ in $\mathbb{F}_p[x]$ is defined in the following manner. For $i = 1, \ldots, \mathfrak{n}$, let $U_i(x) = \mathsf{BRW}(x; M_{1+\delta(i-1)}, \ldots, M_{i\delta})$ and $V(x) = \mathsf{Poly}(x^{2^d}; U_1(x), U_2(x), \ldots, U_{\mathfrak{n}}(x))$. Then

$$P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \;=\; x \cdot \mathsf{Poly}(x; V(x), M_{\delta\mathfrak{n}+1}, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)). \quad (10.9)$$

The idea in d-2LHash is to divide the input sequence of blocks into groups of δ blocks, process each such block using BRW with key τ to obtain $\mathfrak{n}$ outputs and then combine these $\mathfrak{n}$ outputs using Poly with x substituted by $\gamma = \tau^{2^d}$. Finally the output of the Poly call, the left over blocks, and the length block are combined using another Poly call with x substituted by τ. This can be seen as two-level hashing. For binary extension fields, a similar two-level hash function was proposed in [32].

The choice of $\delta = 2^d - 1$ is motivated by the fact that $\mathfrak{d}(\delta) = \delta$, i.e. the degree of BRW on δ blocks is δ (see the second point of Theorem 1). A consequence of this is that the coefficients of the $U_i(x)$ are also the coefficients of $V(x)$. In more details, if we write $U_i(x) = x^\delta + u_{i,\delta-1}x^{\delta-1} + \cdots + u_{i,1}x + u_{i,0}$, then noting that $x^{2^d} = x^{\delta+1}$, we have

$$\begin{aligned} V(x) \;=\; & u_{\mathfrak{n},0} + u_{\mathfrak{n},1}x + \cdots + u_{\mathfrak{n},\delta-1}x^{\delta-1} + x^\delta \\ & + x^{\delta+1}\left(u_{\mathfrak{n}-1,0} + u_{\mathfrak{n}-1,1}x + \cdots + u_{\mathfrak{n}-1,\delta-1}x^{\delta-1} + x^\delta\right) \\ & + x^{2(\delta+1)}\left(u_{\mathfrak{n}-2,0} + u_{\mathfrak{n}-2,1}x + \cdots + u_{\mathfrak{n}-2,\delta-1}x^{\delta-1} + x^\delta\right) \\ & + \cdots \\ & + x^{(\mathfrak{n}-1)(\delta+1)}\left(u_{1,0} + u_{1,1}x + \cdots + u_{1,\delta-1}x^{\delta-1} + x^\delta\right). \end{aligned} \quad (10.10)$$

Remark 12.

1. *When $M_1, \ldots, M_\ell$ are clear from the context, instead of $P_1(x; M_1, \ldots, M_\ell)$ we will write $P_1(x)$. Similarly, when $M_1, \ldots, M_\ell$ and $\mathsf{bin}_{m-1}(L)$ are clear from the context, we will write $P_2(x)$, $P_3(x)$ and $P_4(x)$ instead of $P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$, $P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ and $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ respectively.*

2. *The motivation for considering t-BRWHash and d-2LHash is to avoid an implementation difficulty with* BRWHash. *See Section 10.4.4 for a discussion on this issue.*

10.2.1 Combining hash functions

Timing results show that for messages with a small number of blocks polyHash is faster than the other three hash functions, while for larger number of blocks, either t-BRWHash or d-2LHash is faster. If we use one of the hash functions, then either the performance for short messages is sub-optimal, or the performance for long messages is sub-optimal. In this section, we describe a construction which allows obtaining the best of both the cases.

Let $d \geq 2$ be a positive integer. Let X be a binary string of length $L \geq 0$. Let $(X_1, \ldots, X_\ell)$ be the output of $\mathsf{format}(X)$. We define

$$d\text{-}\mathsf{Hash}_\tau(X) \;=\; \begin{cases} \mathsf{polyHash}_\tau(X) & \text{if } \ell < 2^d; \\ d\text{-}\mathsf{2LHash}_\tau(X) & \text{if } \ell \geq 2^d. \end{cases} \tag{10.11}$$

Note that polyHash uses pad1 while d-2LHash uses pad2. So if $\ell < 2^d$, then pad1 is used on $(X_1, \ldots, X_\ell)$ and if $\ell \geq 2^d$, then pad2 is used on $(X_1, \ldots, X_\ell)$. We later show that this combination of polyHash and d-2LHash produces a secure hash function. In (10.11), it is possible to replace d-2LHash with either BRWHash or d-BRWHash (i.e. t-BRWHash with the parameter t equal to d) and still obtain a secure hash function.

Remark 13. *The hash functions have the design parameters t and d. See Section 10.4.5 for a discussion on these parameters as well as other implementation parameters.*

10.2.2 Naming convention

We adopt the following naming convention. For each of the hash functions, there are two possible sets of parameters in Table 10.2. The choice of the prime p determines the values of m, k and n. So for each of the hash functions, by specifying the value of p, we obtain two different instantiations. If p is chosen to be $2^{127} - 1$, we append 1271 to the name of the hash function, and if p is chosen to be $2^{130} - 5$, we append 1305 to the name of the hash function. For example, polyHash1271 denotes polyHash computed modulo $2^{127} - 1$ and BRWHash1305 denotes BRWHash computed modulo $2^{130} - 5$. The naming convention will become important in Section 10.7 where we provide explicit timings for the various hash functions.

Our naming convention distinguishes between Poly which is used to denote the polynomial in (10.1) and the hash function polyHash given by (10.2) built from Poly after formatting and

padding the message. We require this distinction since we use Poly as a component in the other hash functions. As a result of this distinction, the naming of polyHash1305 is different from Poly1305, even though the two hash functions are identical for messages whose lengths in bits are multiples of eight.

10.3 AXU bounds

The following result will be useful for obtaining AXU bounds for the four hash functions. This result is a generalisation of an observation used in the proof of Theorem 3.3 in [16].

Lemma 1. *Let $p = 2^m - \delta$ be a prime and k be a positive integer such that $k < m$ and $\delta < 2^k - 1$. (The values of p, m and k given in Table 10.2 satisfy these conditions.) Let $\alpha \in \mathbb{Z}_{2^k}$, and $P(x)$ and $P'(x)$ be distinct polynomials in $\mathbb{F}_p[x]$ satisfying $P(0) = P'(0) = 0$. The number of distinct $\tau \in \mathbb{F}_p$ such that*

$$((P(\tau) \bmod p) \bmod 2^k) - ((P'(\tau) \bmod p) \bmod 2^k) \equiv \alpha \pmod{2^k} \tag{10.12}$$

is at most 2^{m-k+1} times the degree of the polynomial $P(x) - P'(x)$.

Consequently, for τ chosen uniformly at random from $\{0,1\}^k$ (which is considered to be a subset of $\mathbb{F}_p$), the probability that (10.12) holds is at most $2^{m-2k+1} \cdot \mathsf{deg}(P(x) - P'(x))$.

Proof. Let U be the set of integers in the interval $[-2^m, 2^m - 1]$ which are congruent to α modulo 2^k. Then $i \cdot 2^k + \alpha$ is in U if and only if $-2^m \le i \cdot 2^k + \alpha \le 2^m - 1$, or equivalently $-2^{m-k} - \alpha/2^k \le i \le 2^{m-k} - (\alpha+1)/2^k$. Since $\alpha \in \mathbb{Z}_{2^k}$, the values that i can take are $-2^{m-k}, \ldots, 2^{m-k} - 1$ and hence $\#U = 2^{m-k+1}$.

Suppose (10.12) holds for some $\tau \in \mathbb{F}_p$. Note that $P(\tau) \bmod p$ and $P'(\tau) \bmod p$ are integers in the interval $[0, p-1]$. Write $(P(\tau) \bmod p) = a_1 2^k + a_0$ where $0 \le a_0 < 2^k$ and

$$0 \le a_1 = \lfloor (P(\tau) \bmod p)/2^k \rfloor \le \lfloor (p-1)/2^k \rfloor = \lfloor (2^m - (\delta+1))/2^k \rfloor = 2^{m-k} - 1.$$

The last equality holds since $\delta < 2^k - 1$. Similarly, write $(P'(\tau) \bmod p) \bmod 2^k = b_0$, where $0 \le b_0 < 2^k$ and $0 \le b_1 \le 2^{m-k} - 1$. From the bounds on a_1 and b_1, it follows that

$$-2^{m-k} \le a_1 - b_1, a_1 - b_1 - 1 \le 2^{m-k} - 1.$$

Since τ satisfies (10.12), we have $a_0 - b_0 \equiv \alpha \bmod 2^k$. So over the integers either $a_0 - b_0 = \alpha$ or $a_0 - b_0 = -2^k + \alpha$ according as $a_0 \ge b_0$ or $a_0 < b_0$ respectively. Let $\beta = 2^k(a_1 - b_1) + \alpha$ if $a_0 \ge b_0$, and $\beta = 2^k(a_1 - b_1 - 1) + \alpha$ if $a_0 < b_0$. Then $\beta \equiv \alpha \bmod 2^k$ and from the above bounds on $a_1 - b_1$ and $a_1 - b_1 - 1$, we have $\beta \in U$. So it follows that if τ satisfies (10.12), then $(P(\tau) \bmod p) - (P'(\tau) \bmod p) = 2^k(a_1 - b_1) + a_0 - b_0$ which is equal to $2^k(a_1 - b_1) + \alpha$ if $a_0 \ge b_0$ and is equal to $2^k(a_1 - b_1 - 1) + \alpha$ if $a_0 < b_0$. So $(P(\tau) \bmod p) - (P'(\tau) \bmod p) = \beta$. Then τ is a root of the polynomial $P(x) - P'(x) - \beta \pmod p$.

Let $R_\beta(x) = P(x) - P'(x) - \beta \in \mathbb{F}_p[x]$. Since $P(x)$ and $P'(x)$ are distinct polynomials and $P(0) = P'(0) = 0$, we have $R_\beta(x) \in \mathbb{F}_p[x]$ to be a non-zero polynomial whose degree is at most the degree of $P(x) - P'(x)$. Since $\beta \in U$, $\#U = 2^{m-k+1}$ and the number of roots of $R_\beta(x)$ over $\mathbb{F}_p$ is at most the degree of $R_\beta(x)$, it follows that the number of distinct τ such that (10.12) holds is at most 2^{m-k+1} times the degree of $R_\beta(x)$. □

Lemma 1 reduces the problem of determining the probability that a uniform random k-bit string τ satisfies (10.12) to the simpler problem of determining the degree of the polynomial $P(x) - P'(x) \in \mathbb{F}_p[x]$.

Lemma 2. *Let X be a binary string of length $L > 0$. Let $\ell = \lceil L/n \rceil$ and suppose $(M_1, \ldots, M_\ell)$ is the output of $\mathsf{pad1}(\mathsf{format}(X))$. Then $M_i \neq 0^{m-1}$ for $i = 1, \ldots, \ell$.*

Lemma 3. *Let X be a binary string of length $L \geq 0$. Then the maps $X \mapsto \mathsf{pad1}(\mathsf{format}(X))$ and $X \mapsto \mathsf{pad2}(\mathsf{format}(X))$ are injections.*

Proof. Let X and X' be two distinct binary strings of lengths L and L' respectively and we assume without loss of generality that $L \geq L' \geq 0$. Let $\ell = \lceil L/n \rceil$ and $\ell' = \lceil L'/n \rceil$. Let $(X_1, \ldots, X_\ell)$ be the output of $\mathsf{format}(X)$ and $(X'_1, \ldots, X'_{\ell'})$ be the output of $\mathsf{format}(X')$. Let s and s' be the lengths of X_ℓ and $X'_{\ell'}$ respectively.

We first consider $\mathsf{pad1}(\mathsf{format}(X))$. Suppose $(M_1, \ldots, M_\ell)$ is the output of $\mathsf{pad1}(\mathsf{format}(X))$ and $(M'_1, \ldots, M'_{\ell'})$ is the output of $\mathsf{pad1}(\mathsf{format}(X'))$. If $\ell \neq \ell'$, then clearly $(M_1, \ldots, M_\ell) \neq (M'_1, \ldots, M'_{\ell'})$.

So suppose that $\ell = \ell'$. Since $X \neq X'$, there must be an i in $\{1, \ldots, \ell\}$ such that $X_i \neq X'_i$. If $1 \leq i \leq \ell - 1$, then both X_i and X'_i are full blocks and then $M_i = 0^{m-n-2}||1||X_i \neq 0^{m-n-2}||1||X'_i = M'_i$. So suppose that $i = \ell$. Then $M_\ell = 0^{m-s-2}||1||X_\ell$ and $M'_\ell = 0^{m-s'-2}||1||X'_\ell$. If $s \neq s'$, then the positions of the leading 1 in M_ℓ and M'_ℓ are different and so $M_\ell \neq M'_\ell$; on the other hand, if $s = s'$, then X_ℓ and X'_ℓ have the same length and since they are distinct, there must be a bit position where they differ, in which case we again have $M_\ell \neq M'_\ell$.

Next let us consider $\mathsf{pad2}(\mathsf{format}(X))$. Let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$ and $(M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L'))$ be the output of $\mathsf{pad2}(\mathsf{format}(X'))$. If $\ell \neq \ell'$, then the number of components in the two outputs are different and so the outputs are different. If $\ell = \ell'$ but $L \neq L'$, then $\mathsf{bin}_{m-1}(L) \neq \mathsf{bin}_{m-1}(L')$ and again the two outputs are different. So suppose $L = L'$ which implies $s = s'$. Since $X \neq X'$, there must be an i in $\{1, \ldots, \ell\}$ such that $X_i \neq X'_i$. If $1 \leq i \leq \ell - 1$, then $M_i = 0^{m-n-1}||X_i \neq 0^{m-n-1}||X'_i = M'_i$ and if $i = \ell$, then $M_\ell = 0^{m-s-1}||X_\ell \neq 0^{m-s-1}||X'_\ell = M'_\ell$. □

Lemma 4. *Let X be a binary string of length $L \geq 0$. Let $\ell = \lceil L/n \rceil$.*

1. *Suppose $(M_1, \ldots, M_\ell)$ is the output of $\mathsf{pad1}(\mathsf{format}(X))$. Then*

$$X \;\mapsto\; P_1(x; M_1, \ldots, M_\ell)$$

   is an injection.

2. *Suppose $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is the output of $\mathsf{pad2}(\mathsf{format}(X))$. Then the maps*

$$\begin{aligned} X &\mapsto P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)), \\ X &\mapsto P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)), \\ X &\mapsto P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \end{aligned}$$

   are injections.

Proof. Let X and X' be two distinct binary strings of lengths L and L' respectively and we assume without loss of generality that $L \geq L' \geq 0$. Let $\ell = \lceil L/n \rceil$ and $\ell' = \lceil L'/n \rceil$.

Suppose $(M_1, \ldots, M_\ell)$ is the output of $\mathsf{pad1}(\mathsf{format}(X))$ and $(M'_1, \ldots, M'_{\ell'})$ is the output of $\mathsf{pad1}(\mathsf{format}(X'))$. From Lemma 3, we have $(M_1, \ldots, M_\ell) \neq (M'_1, \ldots, M'_{\ell'})$ and from Lemma 2, we have $M_i \neq 0^{m-1}$ and $M'_j \neq 0^{m-1}$ for $i = 1, \ldots, \ell$ and $j = 1, \ldots, \ell'$. If $L' = 0$, i.e. X' is the empty string, then X is not the empty string and we have $L > 0$ and so $\ell > 0$. Since $M_1 \neq 0^{m-1}$ it follows that $P_1(x; M_1, \ldots, M_\ell)$ is a non-zero polynomial whereas $P_1(x; M'_1, \ldots, M'_{\ell'}) = P_1(x;) = 0$. Now suppose that $L' > 0$ and so $\ell \geq \ell' \geq 1$. We have $P_1(x; M_1, \ldots, M_\ell) = M_1 x^\ell + \cdots + M_{\ell-1} x^2 + M_\ell x$ and $P_1(x; M'_1, \ldots, M'_{\ell'}) = M'_1 x^\ell + \cdots + M'_{\ell'-1} x^2 + M'_{\ell'} x$. If $\ell > \ell'$, then since $M_1 \neq 0^{m-1}$, it follows that $P_1(x; M_1, \ldots, M_\ell) \neq P_1(x; M'_1, \ldots, M'_{\ell'})$. If $\ell = \ell'$, since $(M_1, \ldots, M_\ell) \neq (M'_1, \ldots, M'_{\ell'})$, it again follows that $P_1(x; M_1, \ldots, M_\ell) \neq P_1(x; M'_1, \ldots, M'_{\ell'})$.

Now we turn to the other three maps. Suppose $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is the output of $\mathsf{pad2}(\mathsf{format}(X))$ and $(M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L'))$ is the output of $\mathsf{pad2}(\mathsf{format}(X'))$. From Lemma 3, we have $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \neq (M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L'))$.

Please note that $\mathsf{bin}_{m-1}(L)$ is the coefficient of x in $P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$, $P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ and $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$. On the other hand, $\mathsf{bin}_{m-1}(L')$ is the coefficient of x in $P_2(x; M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L'))$, $P_3(x; M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L'))$ and $P_4(x; M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L'))$. If $\mathsf{bin}_{m-1}(L) \neq \mathsf{bin}_{m-1}(L')$, then clearly,

$$\begin{aligned}
P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) &\neq P_2(x; M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L')), \\
P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) &\neq P_3(x; M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L')), \\
P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) &\neq P_4(x; M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L')).
\end{aligned}$$

So henceforth suppose that $\mathsf{bin}_{m-1}(L) = \mathsf{bin}_{m-1}(L')$, i.e. $L = L'$ and so $\ell = \ell'$. Since $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is not equal to $(M'_1, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L))$, it follows that $(M_1, \ldots, M_\ell) \neq (M'_1, \ldots, M'_\ell)$. From this point, the arguments for $P_2(x)$, $P_3(x)$ and $P_4(x)$ are different.

Consider $P_2(x)$. Using the injectivity of BRW (see the first point of Theorem 1), we have

$$\begin{aligned}
P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) &= x(x \cdot \mathsf{BRW}(x; M_1, \ldots, M_\ell) + \mathsf{bin}_{m-1}(L)) \\
&\neq x(x \cdot \mathsf{BRW}(x; M'_1, \ldots, M'_\ell) + \mathsf{bin}_{m-1}(L)) \\
&= P_2(x; M'_1, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L)).
\end{aligned}$$

Next consider $P_3(x)$. Since $\ell = \ell'$, it follows that $\mathfrak{m} = \mathfrak{m}'$. Using the injectivity of BRW and Poly, we have

$$\begin{aligned}
&P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \\
&= x \cdot \mathsf{Poly}(x; \mathsf{BRW}(x; M_1, M_2, \ldots, M_\mathfrak{m}), M_{\mathfrak{m}+1}, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \\
&\neq x \cdot \mathsf{Poly}(x; \mathsf{BRW}(x; M'_1, M'_2, \ldots, M'_\mathfrak{m}), M'_{\mathfrak{m}+1}, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L)) \\
&= P_3(x; M'_1, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L)).
\end{aligned}$$

Finally consider $P_4(x)$. Since $\ell = \ell'$, it follows that $\mathfrak{n} = \mathfrak{n}'$. If $(M_{1+\delta(i-1)}, \ldots, M_{i\delta}) \neq (M'_{1+\delta(i-1)}, \ldots, M'_{i\delta})$ for some i in $\{1, \ldots, \mathfrak{n}\}$, then by the injectivity of BRW,

$$U_i(x) = \mathsf{BRW}(x; M_{1+\delta(i-1)}, \ldots, M_{i\delta}) \neq \mathsf{BRW}(x; M'_{1+\delta(i-1)}, \ldots, M'_{i\delta}) = U'_i(x).$$

By construction, the coefficients of $V(x)$ are the coefficients of the U_i's (see (10.10)) and so $U_i(x) \neq U'_i(x)$ implies that $V(x) \neq V'(x)$. On the other hand, if $(M_{1+\delta(i-1)}, \ldots, M_{i\delta}) = (M'_{1+\delta(i-1)}, \ldots, M'_{i\delta})$ for all i in $\{1, \ldots, \mathfrak{n}\}$, then since $(M_1, \ldots, M_\ell) \neq (M'_1, \ldots, M'_\ell)$, it follows that $(M_{\delta\mathfrak{n}+1}, \ldots, M_\ell) \neq (M'_{\delta\mathfrak{n}+1}, \ldots, M'_\ell)$. In either case, we have

$$\begin{aligned} & P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \\ &= \mathsf{Poly}(x; V(x), M_{\delta\mathfrak{n}+1}, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \\ &\neq \mathsf{Poly}(x; V'(x), M'_{\delta\mathfrak{n}+1}, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L)) \\ &= P_4(x; M'_1, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L)). \end{aligned}$$

□

Lemma 5. *Let X be a binary string of length $L \geq 1$ and n be a positive integer. Let $\ell = \lceil L/n \rceil$.*

1. *Let $(M_1, \ldots, M_\ell)$ be the output of $\mathsf{pad1}(\mathsf{format}(X))$. Then $\mathsf{deg}(P_1(x; M_1, \ldots, M_\ell)) = \ell$.*
2. *Let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$.*
   (a) *Then $\mathsf{deg}(P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))) \leq 1 + 2\ell$.*
   (b) *Let $t \geq 2$ be the parameter of t-$\mathsf{BRWHash}$. Then*

$$\mathsf{deg}(P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))) \leq 1 + 2\ell - (\ell \bmod 2^t) \leq 1 + 2\ell.$$

   (c) *Let $d \geq 2$ be the parameter of d-$\mathsf{2LHash}$. Then*

$$\mathsf{deg}(P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))) \leq 1 + (2^d/(2^d - 1))\ell \leq 1 + 2\ell.$$

Proof. Since $L \geq 1$, we have $\ell \geq 1$.

Since $\ell \geq 1$, using the definition of Poly, we have $P_1(x; M_1, \ldots, M_\ell) = M_1 x^\ell + M_2 x^{\ell-1} + \cdots + M_\ell x$. By Lemma 2, M_1 is a non-zero element of $\mathbb{F}_p$. So the degree of $P_1(x)$ is ℓ.

Let $\rho = \mathfrak{d}(\ell)$ and $a_0, \ldots, a_\rho$ be such that $\mathsf{BRW}(x; M_1, \ldots, M_\ell) = a_\rho x^\rho + \cdots + a_1 x + a_0$. (Note that if $\ell \geq 3$, then since $\mathsf{BRW}(x; M_1, \ldots, M_\ell)$ is monic, it follows that $a_\rho = 1$.) So from (10.5),

$$\begin{aligned} P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) &= x(x \cdot \mathsf{BRW}(x; M_1, \ldots, M_\ell) + \mathsf{bin}_{m-1}(L)) \\ &= a_\rho x^{\rho+2} + \cdots + a_1 x^3 + a_0 x^2 + \mathsf{bin}_{m-1}(L) x. \end{aligned}$$

From the second point of Theorem 1, we have $\rho \leq 2\ell - 1$ and so the degree of $P_2(x)$ is at most $2\ell - 1 + 2 = 2\ell + 1$.

Let $\mathfrak{m} = \ell - (\ell \bmod 2^t)$. Let $\rho = \mathfrak{d}(\mathfrak{m})$ and $a_0, \ldots, a_\rho$ be such that $\mathsf{BRW}(x; M_1, \ldots, M_\mathfrak{m}) = a_\rho x^\rho + \cdots + a_1 x + a_0$. (As above, if $\mathfrak{m} \geq 3$, then $a_\rho = 1$.) So

$$\begin{aligned} P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) &= x \cdot \mathsf{Poly}(x; \mathsf{BRW}(x; M_1, M_2, \ldots, M_\mathfrak{m}), M_{\mathfrak{m}+1}, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \\ &= a_\rho x^{\ell-\mathfrak{m}+2+\rho} + \cdots + a_1 x^{\ell-\mathfrak{m}+3} + a_0 x^{\ell-\mathfrak{m}+2} \\ &\quad + M_{\mathfrak{m}+1} x^{\ell-\mathfrak{m}+1} + \cdots + M_\ell x^2 + \mathsf{bin}_{m-1}(L) x. \end{aligned}$$

The degree of $P_3(x)$ is at most $\ell - \mathfrak{m} + 2 + \rho$. From the second point of Theorem 1, $\rho \leq 2\mathfrak{m} - 1$. Using this in the expression for the degree and noting that $\mathfrak{m} \leq \ell$ gives us the desired bound.

Let $\delta = 2^d - 1 \geq 3$ and $\mathfrak{n} = \lfloor \ell/\delta \rfloor$. Recall that $U_i(x) = \mathsf{BRW}(x; M_{1+\delta(i-1)}, \ldots, M_{i\delta})$. Note that $\delta \geq 3$ implies that each $U_i(x)$ is a monic polynomial. Further, since $\delta = 2^d - 1$, from the second point of Theorem 1, we have $\mathsf{deg}(U_i) = \delta$. Since $V(x) = \mathsf{Poly}(x^{2^d}; U_1(x), \ldots, U_{\mathfrak{n}}(x))$, the coefficients of $V(x)$ are the coefficients of the $U_i(x)$'s and $\mathsf{deg}(V) = (\delta+1)(\mathfrak{n}-1) + \delta = (\delta+1)\mathfrak{n} - 1$ (see (10.10)). From (10.9) the degree of $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is equal to $\mathsf{deg}(V) + \ell - \delta\mathfrak{n} + 2 = \mathfrak{n} + \ell + 1$. Since $\mathfrak{n} \leq \ell/\delta$, we obtain the desired bound on the degree of $P_4(x)$. □

We obtain the following simple lower bound on the degrees of P_2, P_3 and P_4 which will be required in arguing about the correctness of d-Hash given by (10.11).

Corollary 1. *Let $d \geq 2$ and n be positive integers. Let X be a binary string of length $L \geq 1$ such that $\ell = \lceil L/n \rceil \geq 2^d$. Let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$. Let the parameter t of t-$\mathsf{BRWHash}$ be equal to d. Then $P_2(x)$, $P_3(x)$ and $P_4(x)$ are monic polynomials having degrees at least $2^d + 2$.*

Proof. Since $d \geq 2$ we have $\ell \geq 2^d > 3$. So the BRW components of all the three polynomials P_2, P_3 and P_4 are for more than three blocks and from the definition of BRW polynomials, it follows that these BRW components are all monic. Using the definitions of P_2, P_3 and P_4, it follows that all the three polynomials are also monic.

Using $\ell \geq 2^d > 3$, from the second point of Theorem 1 and the proof of Lemma 5 we have the following.

1. $\mathsf{deg}(P_2) = \mathfrak{d}(\ell) + 2 = (2^{\lfloor \lg \ell \rfloor + 1} - 1) + 2 \geq 2^d + 2$.
2. $\mathfrak{m} = \ell - (\ell \bmod 2^d) \geq 2^d$ and so $\mathsf{deg}(P_3) = \ell - \mathfrak{m} + 2 + \mathfrak{d}(\mathfrak{m}) = (\ell \bmod 2^t) + \mathfrak{d}(m) + 2 = (\ell \bmod 2^t) + (2^{\lfloor \lg \mathfrak{m} \rfloor + 1} - 1) + 2 \geq 2^d + 2$.
3. $\mathfrak{n} = \lfloor \ell/(2^d - 1) \rfloor \geq 1$ and so $\mathsf{deg}(P_4) = \mathfrak{n} + \ell + 1 \geq 2^d + 2$.

□

Theorem 2. *Suppose the prime p and the parameters m, k and n are as defined in Table 10.2. Let X and X' be two distinct binary strings of lengths L and L' respectively with $L \geq L' \geq 0$, and α be an element of $\mathbb{Z}_{2^k}$. Let $\ell = \lceil L/n \rceil$. Suppose τ is chosen uniformly at random from $\{0,1\}^k$. Let $t \geq 2$ be the parameter of t-$\mathsf{BRWHash}$ and $d \geq 2$ be the parameter of d-$\mathsf{2LHash}$. Then*

$$\begin{aligned}
\Pr[\mathsf{polyHash}_\tau(X) - \mathsf{polyHash}_\tau(X') = \alpha] &\leq \ell \cdot 2^{m-2k+1}, \\
\Pr[\mathsf{BRWHash}_\tau(X) - \mathsf{BRWHash}_\tau(X') = \alpha] &\leq (1 + 2\ell) \cdot 2^{m-2k+1}, \\
\Pr[t\text{-}\mathsf{BRWHash}_\tau(X) - t\text{-}\mathsf{BRWHash}_\tau(X') = \alpha] &\leq (1 + 2\ell) \cdot 2^{m-2k+1}, \\
\Pr[d\text{-}\mathsf{2LHash}_\tau(X) - d\text{-}\mathsf{2LHash}_\tau(X') = \alpha] &\leq (1 + 2\ell) \cdot 2^{m-2k+1}.
\end{aligned}$$

Proof. Lemma 1 reduces the problem of upper bounding the differential probabilities of the hash functions to the analysis of the polynomials over $\mathbb{F}_p$ which define the corresponding hash functions. Specifically, we need to show that the constant terms of these polynomials are 0 and distinct X and X' map to distinct polynomials. From the definitions of $P_1(x)$, $P_2(x)$, $P_3(x)$ and $P_4(x)$ it follows that the constant terms of these polynomials are 0. The distinctness of the polynomials corresponding to distinct X and X' is given by Lemma 4. So Lemma 1 can be applied. Using the expressions for the degrees of the relevant polynomials from Lemma 5, we obtain the desired bounds on the probabilities. □

We next show the AXU bound for d-Hash.

Theorem 3. *Suppose the prime p and the parameters m, k and n are as defined in Table 10.2. Let X and X' be two distinct binary strings of lengths L and L' respectively with $L \geq L' \geq 0$, and α be an element of $\mathbb{Z}_{2^k}$. Let $\ell = \lceil L/n \rceil$. Suppose τ is chosen uniformly at random from $\{0,1\}^k$. Then*

$$\Pr[d\text{-}\mathsf{Hash}_\tau(X) - d\text{-}\mathsf{Hash}_\tau(X') = \alpha] \;\leq\; (1+2\ell)\cdot 2^{m-2k+1}. \tag{10.13}$$

Proof. By construction, d-Hash applies $\mathsf{polyHash}$ if the number of blocks is less than 2^d and applies d-$\mathsf{2LHash}$ if the number of blocks is at least 2^d. Let $\ell' = \lceil L'/n \rceil$. There are three cases to consider, namely $\ell, \ell' < 2^d$, $\ell, \ell' \geq 2^d$, and $\ell' < 2^d \leq \ell$.

First suppose $\ell, \ell' < 2^d$. In this case, for both X and X', $\mathsf{polyHash}$ is applied and using Theorem 2 we have,

$$\begin{aligned}\Pr[d\text{-}\mathsf{Hash}_\tau(X) - d\text{-}\mathsf{Hash}_\tau(X') = \alpha] &= \Pr[\mathsf{polyHash}_\tau(X) - \mathsf{polyHash}_\tau(X') = \alpha] \\ &\leq \ell \cdot 2^{m-2k+1} < (1+2\ell)\cdot 2^{m-2k+1}.\end{aligned}$$

Next suppose $\ell, \ell' \geq 2^d$. In this case, for both X and X', d-$\mathsf{2LHash}$ is applied and using Theorem 2 we have

$$\begin{aligned}\Pr[d\text{-}\mathsf{Hash}_\tau(X) - d\text{-}\mathsf{Hash}_\tau(X') = \alpha] &= \Pr[d\text{-}\mathsf{2LHash}_\tau(X) - d\text{-}\mathsf{2LHash}_\tau(X') = \alpha] \\ &\leq (1+2\ell)\cdot 2^{m-2k+1}.\end{aligned}$$

Now suppose $\ell' < 2^d \leq \ell$. In this case, X is hashed with d-$\mathsf{2LHash}$ and X' is hashed with $\mathsf{polyHash}$. Recall that d-$\mathsf{2LHash}$ processes X using $\mathsf{pad2}$ while $\mathsf{polyHash}$ processes X' using $\mathsf{pad1}$. Let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$ and $(M'_1, \ldots, M'_{\ell'})$ be the output of $\mathsf{pad1}(\mathsf{format}(X'))$. The constant terms of both $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ and $P_1(x; M'_1, \ldots, M'_{\ell'})$ are zero. From (10.2) and Lemma 2, the degree of $P_1(x; M'_1, \ldots, M'_{\ell'})$ is equal to $\ell' < 2^d$. Since $\ell \geq 2^d$, by Corollary 1, $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is a monic polynomial of degree at least $2^d + 2$. So $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ and $P_1(x; M'_1, \ldots, M'_{\ell'})$ are distinct polynomials whose constant terms are zero. Applying Lemma 1 to these two polynomials, we have $\Pr[d\text{-}\mathsf{Hash}_\tau(X) - d\text{-}\mathsf{Hash}_\tau(X') = \alpha]$ to be at most 2^{m-2k+1} times the degree of $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) - P_1(x; M'_1, \ldots, M'_{\ell'})$ which is the degree of $P_4(x)$ and by Lemma 5 this degree is at most $1 + 2\ell$. □

Table 10.3: For the two primes in Table 10.2, the values of ϵ such that the hash families are ϵ-AXU. Here ℓ is the number of message blocks.

	polyHash	BRWHash, t-BRWHash, d-2LHash, d-Hash
$2^{127}-1$	$\ell \cdot 2^{-125}$	$(2\ell+1)2^{-125}$
$2^{130}-5$	$\ell \cdot 2^{-124}$	$(2\ell+1)2^{-124}$

Remark 14. *Suppose that in* d-Hash*, the hash function* d-2LHash *is replaced by either* BRWHash *or* t-BRWHash*. Using the bounds on the degrees of these polynomials given by Lemma 5 and Corollary 1, the proof of Theorem 3 shows that the bound* (10.13) *also holds for such modified variations of* d-Hash*.*

Remark 15. *The probability bounds in Theorems 2 and 3 are in terms of the number* ℓ *of* n*-bit blocks. So setting an upper bound on* ℓ *provides the values of* ϵ *for the hash families to be* ϵ*-AXU. Suppose we set* $\ell = 2^{48}$*, i.e. we restrict the hash families to process messages having at most* 2^{48} *blocks. The corresponding values of* ϵ *for the four hash families and the two primes can be obtained from the expressions given in Table 10.3 and are at most* 2^{-75}*. Choosing a lower value of* ℓ*, will further decrease the value of* ϵ*.*

With $\ell \leq 2^{48}$*, we have* $L \leq 2^{55}$ *for the values of* n *in Table 10.2. Since it is unlikely that there will be an application which will require to hash messages longer than* 2^{55} *bits, for practical purposes, the length of any message will fit in a 64-bit word. So in the constructions* BRWHash*,* t-BRWHash *and* d-2LHash*,* $\mathsf{bin}_{m-1}(L)$ *is essentially* $\mathsf{bin}_{64}(L)$*. This provides a speed-up in the implementation of the field multiplication with the binary representation of the length of the message. The effect, however, is minor since there is only one such field multiplication.*

10.3.1 Explanations for the padding schemes

Suppose $(M_1, \ldots, M_\ell)$ is the output of $\mathsf{pad1}(\mathsf{format}(X))$. Lemma 2 states that each of the M_i's are non-zero. The proof of Lemma 4 uses this observation to show that the map $X \mapsto \mathsf{Poly}(x; M_1, \ldots, M_\ell)$ and hence the map $X \mapsto P_1(x; M_1, \ldots, M_\ell)$ are injections.

The fact that the M_i's are non-zero is not sufficient to argue that the map $X \mapsto \mathsf{BRW}(x; M_1, \ldots, M_\ell)$ is an injection. For example, suppose X and X' are distinct binary strings such that $(M_1, \ldots, M_5) = \mathsf{pad1}(\mathsf{format}(X))$ and $(M'_1, \ldots, M'_6) = \mathsf{pad1}(\mathsf{format}(X'))$. Then

$$\begin{aligned} \mathsf{BRW}(x; M_1, \ldots, M_5) &= ((x + M_1)(x^2 + M_2) + M_3)(x^4 + M_4) + M_5, \\ \mathsf{BRW}(x; M'_1, \ldots, M'_6) &= ((x + M'_1)(x^2 + M'_2) + M'_3)(x^4 + M'_4) + M'_5 x + M'_6. \end{aligned}$$

Note that both $\mathsf{BRW}(x; M_1, \ldots, M_5)$ and $\mathsf{BRW}(x; M'_1, \ldots, M'_6)$ are monic polynomials of degrees equal to 7. Since the coefficients of $\mathsf{BRW}(x; M_1, \ldots, M_5)$ and $\mathsf{BRW}(x; M'_1, \ldots, M'_6)$ are nonlinear functions of $M_1, \ldots, M_5$ and $M'_1, \ldots, M'_6$ respectively, the fact that the M_i's and the M'_j's are non-zero is not sufficient to argue that $\mathsf{BRW}(x; M_1, \ldots, M_5) \neq \mathsf{BRW}(x; M'_1, \ldots, M'_6)$.

Theorem 5.6 of [17] shows that if S is a subset of $\mathbb{F}_p$ such that $S \cap (S+1) = \emptyset$, then BRW injectively maps $\cup_{i \geq 0} S^i$ to $\mathbb{F}_p[x]$. We considered using this result to avoid the above issue. For S we considered taking all elements of $\mathbb{F}_p$ whose least significant bit is 0 (i.e. S is the subset of the even numbers of $\mathbb{F}_p$). Such an S satisfies the requirement $S \cap (S+1) = \emptyset$. The problem arises in defining an appropriate padding scheme. For concreteness, consider the prime $2^{130} - 5$. Suppose $(X_1, \ldots, X_\ell)$ is the output of format(X). Each X_i is an 128-bit string. We may define a padding scheme which left shifts X_i by one place to obtain M_i. The M_i's are ensured to be even and hence are in S. There are two problems to such a padding scheme. For one thing, each X_i is a 16-byte string which would be stored as 2 64-bit or 4 32-bit words. A left shift by one place would require multiple shifts of the words representing an X_i. This would result in some inefficiency. (In comparison, note that appending a 1 on the left, as in pad1, does not require any shift.) There is another more basic problem with such a padding scheme. We need to ensure that a padded partial block is distinct from a padded full block and also that after padding, two partial blocks of different lengths map to distinct strings. We see no simple way of ensuring that a padded partial block is distinct from a padded full block.

Further, the hash functions t-BRWHash and d-2LHash are built using a combination of BRW and Poly. Using a padding scheme to injectivity of only BRW is not sufficient to the injectivity of the defining polynomials for the hash functions t-BRWHash and d-2LHash.

To handle the above problems, we introduced pad2 which includes the binary representation of the length of X in its output. By appropriately using the length block, we obtain injective maps as shown in Lemma 4. The use of pad2 simplifies the description of the three hash functions and also makes the injectivity argument quite easy.

10.4 Algorithms

Algorithms to evaluate the hash functions essentially boil down to evaluating the polynomials $P_1(x)$, $P_2(x)$, $P_3(x)$ and $P_4(x)$ at the point τ. These four polynomials are in turn defined from the two polynomials Poly and BRW. So we first consider algorithms to compute the values of the polynomials Poly and BRW at a point.

In Sections 10.4.1 and 10.4.2, the description of the algorithms to compute the values of Poly and BRW at a point are over an arbitrary finite field $\mathbb{F}$. For $l \geq 0$, let $M_1, \ldots, M_l$ be elements of $\mathbb{F}$ and τ be an element of $\mathbb{F}$. (Recall that ℓ is the number of blocks obtained by formatting a binary string X; l is not necessarily equal to ℓ.) We consider the evaluation of Poly$(x; M_1, \ldots, M_l)$ and BRW$(x; M_1, \ldots, M_l)$ at the point τ. Such evaluation involves multiplications and additions over $\mathbb{F}$. A field multiplication has two steps. In the first step, a multiplication is done over an appropriate structure (such as the ring of integers or polynomials) and in the second step, the product is reduced. By unreducedMult we will denote only the first step, i.e. the reduction step is not performed, while by reduce we will denote the reduction step. Similarly a field addition also has two steps, an addition over an appropriate structure followed by a reduction. In the algorithms for evaluating Poly and BRW, the reduction after addition is never performed. So in the algorithms, '+' denotes an unreduced addition.

10.4.1 Evaluation of Poly

For $l \geq 2$, Using Horner's rule, $\mathsf{Poly}(\tau; M_1, \ldots, M_l)$ can be evaluated as follows.

$$\mathsf{Poly}(\tau; M_1, \ldots, M_l) \;=\; \tau(\cdots\tau(\tau(\tau \cdot M_1 + M_2) + M_3) + \cdots + M_{l-1}) + M_l. \quad (10.14)$$

This requires $l-1$ field multiplications. A delayed (or lazy) reduction strategy was used in [61] to implement the hash function GHASH which is defined over binary fields. It was also used in [60] in the context of evaluation of Poly1305 using vector instructions. The same strategy can also be employed for an arbitrary finite field $\mathbb{F}$. We describe the strategy below.

Let $g \geq 1$ be a parameter. The blocks $M_1, \ldots, M_l$ are divided into $\lceil l/g \rceil$ groups and one reduction will be applied for each group. Let $r \in \{1, \ldots, g\}$ be such that $r \equiv l \bmod g$. Let $k = (l-r)/g$ (so that $k+1 = \lceil l/g \rceil$) and define

$$A_1 \;=\; M_1\tau^{r-1} + M_2\tau^{r-2} + \cdots + M_{r-1}\tau + M_r,$$

and for $i = 1, \ldots, k$,

$$A_{i+1} \;=\; M_{r+(i-1)g+1}\tau^{g-1} + M_{r+(i-1)g+2}\tau^{g-2} + \cdots + M_{r+ig-1}\tau + M_{r+ig}.$$

Then

$$\mathsf{Poly}(\tau; M_1, \ldots, M_l) \;=\; \tau^g(\cdots\tau^g(\tau^g(\tau^g \cdot A_1 + A_2) + A_3) + \cdots + A_k) + A_{k+1}. \quad (10.15)$$

Note that A_1 is defined using a group of $r \leq g$ blocks, while each of $A_2, \ldots, A_{k+1}$ is defined using exactly g blocks.

Suppose the elements $\tau^2, \tau^3, \ldots, \tau^{g-1}, \tau^g$ are computed over $\mathbb{F}$ and stored. Then the A_i's can be evaluated by multiplying the relevant power of g with the appropriate block and summing the products. The A_i's are not individually reduced during the computation as we explain next. Let $\mathsf{unreduced}(A_i)$ denote the computation of A_i without applying the reduction step, i.e. the outputs of all the multiplications and additions in the expression for A_i are kept unreduced.

Let res be a variable which stores the result of the partial computations. Initially the value of res is set to $\mathsf{reduce}(\mathsf{unreduced}(A_1))$. Next, for $i = 1, \ldots, k$, update res to

$$\mathsf{reduce}(\mathsf{unreducedMult}(\mathsf{res}, \tau^g) + \mathsf{unreduced}(A_{i+1})).$$

The final value of res provides the required result.

Computing $\mathsf{unreduced}(A_1)$ requires $r-1$ unreduced multiplications. For $i = 2, \ldots, k+1$, computing $\mathsf{unreduced}(A_i)$ requires $g-1$ unreduced multiplications. Finally, the k updations of res require k unreduced multiplications. So the total number of unreduced multiplications required is $r-1+k(g-1)+k = l-1$. This number is the same as that for evaluating (10.14). The advantage is that for $g > 1$, the number of reductions decreases. From the above description, the number of reductions required to compute (10.15) is $k+1 = \lceil l/g \rceil$. In comparison, computing (10.14) requires $l-1$ reductions.

The trade-off is that the powers $\tau^2, \ldots, \tau^g$ are required to be computed and this requires $g-1$ field multiplications. For short messages, the time to compute the powers of τ will

make the strategy inefficient. To avoid this inefficiency one may pre-compute and store the required powers of τ. There are advantages and disadvantages to both the approaches, i.e. to compute the powers on-the-fly and to precompute and store these powers. In our timing results given later, we provide timings for both the approaches.

The general idea of the delayed reduction strategy suggests that as g increases, the efficiency should improve. This, however, is not true in practice. For efficient execution, it is important that the powers of τ be available in the cache. If the value of g is large, then all the powers of τ cannot be stored in the cache and the efficiency of the method decreases. We have experimented with $g = 4, 8, 16, 32$. For long messages, the choice of $g = 16$ provides the best performance, while for shorter messages, smaller values of g provide better performance.

Remark 16. *The explicit advantage of the delayed reduction strategy is the decrease in the number of reductions. Since the number of unreduced multiplications does not decrease, one does not expect a very sharp increase in efficiency due to the use of delayed reduction. Our experiments, on the other hand, show that using $g = 4$ or $g = 8$ provide a very high jump in speed. (Details of the timing results are given later.) Such a jump cannot be solely explained by the decrease in the number of reductions. We investigated the issue deeply. It turns out that there is a hidden advantage of the delayed reduction strategy. The evaluation of the A_k's require multiplications by the powers of τ. These powers are fixed for the entire duration of the computation. So in effect, one of the operands of the multiplications is a constant. During execution, the powers of τ are kept in the data buffer of the multiplication units of the CPU. It is due to this effect that there is a very significant speed improvement. If we modify the expression where instead of a fixed power of τ, multiplication is done with a variable quantity (which no longer evaluates* Poly*), then the speed drops to the level which would be explained by the decrease in the number of reductions.*

10.4.2 Evaluation of BRW

The definition of BRW is recursive. It is possible to write a recursive program to evaluate BRW. Such a program, however, will be quite inefficient. If the number of blocks is fixed, then it is possible to implement BRW using a straight line code. Some examples are given as follows.

For d-2LHash, we provide examples of straight line codes to compute BRW for various values of d. While these are illustrative, our implementations do not exactly correspond to these straight line codes. Through experimentation, we developed variations which are more efficient in practice though the total number of integer multiplications and reductions remain unchanged.

Case $d = 2$ and $\delta = 3$: The output of $\mathsf{BRW}_\tau(M_1, M_2, M_3)$ is the following expression.

$$(M_1 + \tau)(M_2 + \tau^2) + M_3.$$

A straight line code to compute this expression using one integer multiplication and one reduction is the following.

1. $T_1 \leftarrow M_1 + \tau$; $T_2 \leftarrow M_2 + \tau^2$
2. $T_2 \leftarrow T_1 T_2$
3. $T_2 \leftarrow T_2 + M_3$
4. Output $\mathsf{reduce}(T_2)$

Case $d = 3$ and $\delta = 7$: The output of $\mathsf{BRW}_\tau(M_1, \ldots, M_7)$ is the following expression.

$$((M_1 + \tau)(M_2 + \tau^2) + M_3)(M_4 + \tau^4) + (M_5 + \tau)(M_6 + \tau^2) + M_7.$$

A straight line code to compute this expression using three integer multiplication and two reductions is the following.

1. $T_1 \leftarrow M_1 + \tau$; $T_2 \leftarrow M_2 + \tau^2$; $T_3 \leftarrow M_5 + \tau$; $T_4 \leftarrow M_6 + \tau^2$
2. $T_2 \leftarrow T_1T_2$; $T_4 \leftarrow T_3T_4$
3. $T_2 \leftarrow T_2 + M_3$; $T_4 \leftarrow T_4 + M_7$
4. $T_1 \leftarrow M_4 + \tau^4$
5. $T_2 \leftarrow \mathsf{reduce}(T_2)$; $T_2 \leftarrow T_1T_2$
6. $T_4 \leftarrow T_2 + T_4$
7. Output $\mathsf{reduce}(T_4)$

Case $d = 4$ and $\delta = 15$: The output of $\mathsf{BRW}_\tau(M_1, \ldots, M_{15})$ is the following expression.

$$\begin{aligned}&(((M_1 + \tau)(M_2 + \tau^2) + M_3)(M_4 + \tau^4) + (M_5 + \tau)(M_6 + \tau^2) + M_7)(M_8 + \tau^8)\\ &\quad +((M_9 + \tau)(M_{10} + \tau^2) + M_{11})(M_{12} + \tau^4) + (M_{13} + \tau)(M_{14} + \tau^2) + M_{15}.\end{aligned}$$

A straight line code to compute this expression using seven integer multiplication and four reductions is the following.

1. $T_1 \leftarrow M_1 + \tau$; $T_2 \leftarrow M_2 + \tau^2$; $T_3 \leftarrow M_5 + \tau$; $T_4 \leftarrow M_6 + \tau^2$;
   $T_5 \leftarrow M_9 + \tau$; $T_6 \leftarrow M_{10} + \tau^2$; $T_7 \leftarrow M_{13} + \tau$; $T_8 \leftarrow M_{14} + \tau^2$
2. $T_2 \leftarrow T_1T_2$; $T_4 \leftarrow T_3T_4$; $T_6 \leftarrow T_5T_6$; $T_8 \leftarrow T_7T_8$
3. $T_2 \leftarrow T_2 + M_3$; $T_4 \leftarrow T_4 + M_7$; $T_6 \leftarrow T_6 + M_{11}$; $T_8 \leftarrow T_8 + M_{15}$
4. $T_1 \leftarrow M_4 + \tau^4$; $T_5 \leftarrow M_{12} + \tau^4$;
5. $T_2 \leftarrow \mathsf{reduce}(T_2)$; $T_2 \leftarrow T_1T_2$; $T_6 \leftarrow \mathsf{reduce}(T_6)$; $T_6 \leftarrow T_5T_6$
6. $T_4 \leftarrow T_2 + T_4$; $T_8 \leftarrow T_6 + T_8$
7. $T_3 \leftarrow M_8 + \tau^8$
8. $T_4 \leftarrow \mathsf{reduce}(T_4)$; $T_4 \leftarrow T_3T_4$
9. $T_8 \leftarrow T_4 + T_8$
10. Output $\mathsf{reduce}(T_8)$

Case $d = 5$ and $\delta = 31$: The output of $\mathsf{BRW}_\tau(M_1, \ldots, M_{31})$ is the following expression.

$$\begin{aligned}&((((M_1 + \tau)(M_2 + \tau^2) + M_3)(M_4 + \tau^4) + (M_5 + \tau)(M_6 + \tau^2) + M_7)(M_8 + \tau^8)\\ &\quad +((M_9 + \tau)(M_{10} + \tau^2) + M_{11})(M_{12} + \tau^4) + (M_{13} + \tau)(M_{14} + \tau^2) + M_{15})(M_{16} + \tau^{16})\\ &+(((M_{17} + \tau)(M_{18} + \tau^2) + M_{19})(M_{20} + \tau^4) + (M_{21} + \tau)(M_{22} + \tau^2) + M_{23})(M_{24} + \tau^8)\\ &\quad +((M_{25} + \tau)(M_{26} + \tau^2) + M_{27})(M_{28} + \tau^4) + (M_{29} + \tau)(M_{30} + \tau^2) + M_{31}.\end{aligned}$$

A straight line code to compute this expression using fifteen integer multiplication and eight reductions is the following.

1. $T_1 \leftarrow M_1 + \tau$; $T_2 \leftarrow M_2 + \tau^2$; $T_5 \leftarrow M_5 + \tau$; $T_6 \leftarrow M_6 + \tau^2$
   $T_9 \leftarrow M_9 + \tau$; $T_{10} \leftarrow M_{10} + \tau^2$
2. $T_2 \leftarrow T_1T_2$; $T_6 \leftarrow T_5T_6$; $T_{10} \leftarrow T_9T_{10}$
3. $T_{13} \leftarrow M_{13} + \tau$; $T_{14} \leftarrow M_{14} + \tau^2$; $T_{17} \leftarrow M_{17} + \tau$; $T_{18} \leftarrow M_{18} + \tau^2$
   $T_{21} \leftarrow M_{21} + \tau$; $T_{22} \leftarrow M_{22} + \tau^2$
4. $T_{14} \leftarrow T_{13}T_{14}$; $T_{18} \leftarrow T_{17}T_{18}$; $T_{22} \leftarrow T_{21}T_{22}$
5. $T_{25} \leftarrow M_{25} + \tau$; $T_{26} \leftarrow M_{26} + \tau^2$; $T_{29} \leftarrow M_{29} + \tau$; $T_{30} \leftarrow M_{30} + \tau^2$
   $T_2 \leftarrow T_2 + M_3$; $\mathsf{reduce}(T_2)$; $T_4 \leftarrow M_4 + \tau^4$
6. $T_4 \leftarrow T_2T_4$; $T_{26} \leftarrow T_{25}T_{26}$; $T_{30} \leftarrow T_{29}T_{30}$
7. $T_{10} \leftarrow T_{10} + M_{11}$; $\mathsf{reduce}(T_{10})$; $T_{12} \leftarrow M_{12} + \tau^4$; $T_{20} \leftarrow M_{20} + \tau^4$
   $T_{18} \leftarrow T_{18} + M_{19}$; $\mathsf{reduce}(T_{18})$; $T_8 \leftarrow M_8 + \tau^8$; $T_6 \leftarrow T_6 + M_7$; $T_6 \leftarrow T_4 + T_6$; $\mathsf{reduce}(T_6)$
8. $T_{12} \leftarrow T_{10}T_{12}$; $T_{20} \leftarrow T_{18}T_{20}$; $T_8 \leftarrow T_6T_8$
9. $T_{28} \leftarrow M_{28} + \tau^4$; $T_{26} \leftarrow T_{26} + M_{27}$; $\mathsf{reduce}(T_{26})$; $T_{24} \leftarrow M_{24} + \tau^8$; $T_{22} \leftarrow T_{22} + M_{23}$; $\mathsf{reduce}(T_{22})$
   $T_{16} \leftarrow M_{16} + \tau^{16}$; $T_{14} \leftarrow T_{14} + M_{15}$; $T_{14} \leftarrow T_{12} + T_{14}$; $T_{14} \leftarrow T_8 + T_{14}$; $\mathsf{reduce}(T_{14})$
10. $T_{28} \leftarrow T_{26}T_{28}$; $T_{24} \leftarrow T_{22}T_{24}$; $T_{16} \leftarrow T_{14}T_{16}$
11. $T_{30} \leftarrow T_{30} + M_{31}$; $T_{30} \leftarrow T_{28} + T_{30}$; $T_{30} \leftarrow T_{24} + T_{30}$; $T_{30} \leftarrow T_{16} + T_{30}$
12. Output $\mathsf{reduce}(T_{30})$

A general non-recursive algorithm to evaluate BRW was developed in [59] and it was shown that $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$ can be evaluated using $\lfloor l/2 \rfloor$ unreduced multiplications and $1 + \lfloor l/4 \rfloor$ reductions (plus an additional $\lfloor \lg l \rfloor$ field squarings to compute the required power of τ). This is an improvement over the requirement of $\lfloor l/2 \rfloor$ field multiplications stated in Theorem 1. Following Lemma 1 of [59], the evaluation of $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$ can be written as

$$\mathsf{BRW}(\tau; M_1, \ldots, M_l) \quad = \quad \mathsf{reduce}(\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_l)),$$

where

- $\mathsf{unreducedBRW}(\tau;) = 0$;
- $\mathsf{unreducedBRW}(\tau; M_1) = M_1$;
- $\mathsf{unreducedBRW}(\tau; M_1, M_2) = \mathsf{unreducedMult}(M_1, \tau) + M_2$;
- $\mathsf{unreducedBRW}(\tau; M_1, M_2, M_3) = \mathsf{unreducedMult}((\tau + M_1), (\tau^2 + M_2)) + M_3$;
- $\mathsf{unreducedBRW}(\tau; M_1, M_2, \ldots, M_k)$
  $= \mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_{k-1})), (\tau^k + M_k))$,
  if $k \in \{4, 8, 16, 32, \ldots\}$;
- $\mathsf{unreducedBRW}(\tau; M_1, M_2, \ldots, M_l)$
  $= \mathsf{unreducedBRW}(\tau; M_1, \ldots, M_k) + \mathsf{unreducedBRW}(\tau; M_{k+1}, \ldots, M_l)$,
  if $k \in \{4, 8, 16, 32, \ldots\}$ and $k < l < 2k$.

The idea of the algorithm in [59] is to process groups of 2^t blocks at a time for some integer $t \geq 2$. For each such group, unreducedBRW corresponding to the first $2^t - 1$ blocks is evaluated using a straight line code. Partial results are stored in an array. Depending on the number of blocks that have been processed, some of the partial results are taken from the array and combined with the output of the present iteration and the resulting new partial result is again added to the array. Here we present a variant of the algorithm given in [59]. The variant simplifies the algorithm in [59] by using a different method to store partial results. The number of unreduced multiplications and reductions remain unchanged.

The modified algorithm ComputeBRW is shown in Algorithm 8. The partial results are stored in stack and top points to the top of stack. The stack is implemented as an array $\mathsf{stack}[0, ..., \lfloor \lg l \rfloor - t]$. The operation $\mathsf{ntz}(i)$ in Step 11 returns the number of trailing zeros in i; it can be implemented in assembly using the instruction `tzcnt`. The operation $\mathsf{wt}(\lfloor l/2^t \rfloor)$ in Step 20 returns the Hamming weight of the binary representation of $\lfloor l/2^t \rfloor$; it can be implemented in assembly using the instruction `popcnt`.

Algorithm 8 differs from the algorithm in [59] in the manner in which the partial results are stored. Since the manner of storage determines the overall correctness of the algorithm, the proof of correctness provided in [59] needs substantial modifications to apply to Algorithm 8. The following result states the correctness and complexity of the new algorithm.

Theorem 4. *For any $l \geq 0$ and any $t \geq 2$, Algorithm 8 correctly computes* $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$. *The number of* unreducedMult *required is $\lfloor l/2 \rfloor$ and the number of reductions required is $1 + \lfloor l/4 \rfloor$. Additionally $\lfloor \lg l \rfloor$ field squarings are required to compute the powers of τ. The maximum size of* stack *is at most $\lfloor \lg l \rfloor - t + 1$.*

Algorithm 8 Evaluation of $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$, $l \geq 0$. In the algorithm $t \geq 2$ is a parameter.

```
1: function ComputeBRW(τ, M_1, ..., M_l)
2:     keyPow[0] ← τ
3:     if l > 2 then
4:         for j ← 1 to ⌊lg l⌋ do
5:             keyPow[j] ← keyPow[j − 1]^2
6:         end for
7:     end if
8:     top ← −1
9:     for i ← 1 to ⌊l/2^t⌋ do
10:        tmp ← unreducedBRW(τ; M_{2^t(i−1)+1}, ..., M_{2^t·i−1});
11:        k ← ntz(i)
12:        for j ← 0 to k − 1 do
13:            tmp ← tmp + stack[top]; top ← top − 1
14:        end for
15:        tmp ← unreducedMult(reduce(tmp), M_{2^t·i} + keyPow[t + k])
16:        top ← top + 1; stack[top] ← tmp
17:    end for;
18:    r ← l mod 2^t;
19:    tmp ← unreducedBRW(τ; M_{l−r+1}, ..., M_l);
20:    i ← wt(⌊l/2^t⌋)
21:    for j ← 0 to i − 1 do
22:        tmp ← tmp + stack[top]; top ← top − 1
23:    end for
24:    return reduce(tmp);
25: end function.
```

The parameter t in Algorithm 8 does not have any effect on the correctness of the algorithm or on the number of operations that are required. Step 10 uses a straight line code to compute $\mathsf{unreducedBRW}$ on $2^t - 1$ blocks. So the parameter t determines the extent of loop unrolling. This has an effect on the practical efficiency of implementation as reflected in our timing results given later.

10.4.3 Correctness and complexity of Algorithm 8

We require the following result from [59].

Lemma 6 (Lemma 2 of [59]). *Let $t \geq 2$ be an integer. For any $l \geq 2^t$, write*

$$\left\lfloor \frac{l}{2^t} \right\rfloor \;=\; 2^{k_1} + 2^{k_2} + \cdots + 2^{k_s}, \tag{10.16}$$

where $k_1, \ldots, k_s$ are integers such that $k_1 > k_2 > \cdots > k_s \geq 0$. Let $K_0 = 0$ and for $\jmath = 0, \ldots, s-1$, let $K_{\jmath+1} = K_\jmath + 2^{t+k_{\jmath+1}}$. Then

$$\begin{aligned}
&\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_l) \\
&= \mathsf{unreducedBRW}(\tau; M_{K_0+1}, \ldots, M_{K_1}) + \mathsf{unreducedBRW}(\tau; M_{K_1+1}, \ldots, M_{K_2}) \\
&\quad + \cdots + \mathsf{unreducedBRW}(\tau; M_{K_{s-1}+1}, \ldots, M_{K_s}) + \\
&\qquad + \mathsf{unreducedBRW}(\tau; M_{K_s+1}, \ldots, M_l).
\end{aligned} \tag{10.17}$$

It is easy to see that for $l > 2$, Steps 3 to 7 of Algorithm 8 that $\mathsf{keyPow}[j] = \tau^{2^j}$, for $j = 1, \ldots, \lfloor \lg l \rfloor$. The main result required to argue the correctness of Algorithm 8 is the following which shows that the partial results are correctly computed and stored. This result is the counterpart of Lemma 5 of [59].

Lemma 7. *Let $t \geq 2$ and $l \geq 2^t$. Let the loop counter $i \in \{1, \ldots, i_{\max}\}$, with $i_{\max} = \lfloor l/2^t \rfloor$, in Step 9 of Algorithm 8 be written as*

$$i \;=\; 2^{k_{i,1}} + 2^{k_{i,2}} + \cdots + 2^{k_{i,s_i}} \tag{10.18}$$

where $k_{i,1} > k_{i,2} > \cdots > k_{i,s_i} \geq 0$. Let $K_{i,0} = 0$ and for $\jmath = 0, \ldots, s_i - 1$, let $K_{i,\jmath+1} = K_{i,\jmath} + 2^{t+k_{i,\jmath+1}}$. After i iterations of the loop given by Steps 9 to 17, the following properties hold:

$\mathsf{top} = s_i - 1$, and for $\jmath \in \{0, \ldots, s_i - 1\}$, $\mathsf{stack}[\jmath] = \mathsf{unreducedBRW}(\tau; M_{K_{i,\jmath}+1}, \ldots, M_{K_{i,\jmath+1}})$.

Proof. First note that from the definition of $K_{i,\jmath}$, we have

$$K_{i,0} = 0,\; K_{i,1} = 2^{t+k_{i,1}}, K_{i,2} = 2^{t+k_{i,1}} + 2^{t+k_{i,2}}, \ldots, K_{i,s_i} = 2^{t+k_{i,1}} + \cdots + 2^{t+k_{i,s_i}} = 2^t \cdot i. \tag{10.19}$$

The proof is by induction on $i \geq 1$. It mirrors the proof of Lemma 5 of [59], with modifications to handle the different manner in which the partial results are stored and accessed by Algorithm 8 from the algorithm in [59].

The base case is $i = 1$. In this case, $s_1 = 1$, $k_{1,1} = 0$ and $K_{1,1} = 2^t$. The variable tmp is set to unreducedBRW$(\tau; M_1, \ldots, M_{2^t-1})$; the variable k is set to ntz$(1) = 0$ and so the loop in Steps 12 to 14 is not executed. In Step 15, tmp is updated to unreducedMult(reduce(tmp), M_{2^t}+ keyPow$[t]$). In Step 16, the variable top is incremented to 0 (from -1) and stack$[0]$ is assigned the value of tmp. At the end of the first iteration, the value of top is $0 = s_1 - 1$. The argument for the base case of $i = 1$ will be completed if we are able to show that

$$\text{unreducedMult}(\text{reduce}(\text{unreducedBRW}(\tau; M_1, \ldots, M_{2^t-1})), M_{2^t} + \text{keyPow}[t])$$

equals unreducedBRW$(\tau; M_1, \ldots, M_{2^t})$. This equality follows from the definition of unreducedBRW (see Section 10.4.2).

For the inductive step suppose the result holds for $i = 2^{k_{i,1}} + 2^{k_{i,2}} + \cdots + 2^{k_{i,s_i}} \geq 1$. We show that the result holds for $i + 1$. We have

$$i + 1 \;=\; 2^{k_{i,1}} + 2^{k_{i,2}} + \cdots + 2^{k_{i,s_i}} + 1 = 2^{k_{i+1,1}} + 2^{k_{i+1,2}} + \cdots + 2^{k_{i+1,s_{i+1}}}.$$

Note that s_{i+1} can be smaller than s_i. For example, if $i = 11 = 2^3 + 2 + 1$, then $s_{11} = 3$, and $i + 1 = 12 = 2^3 + 2^2$ with $s_{12} = 2$. Such a situation arises when i is odd. So the proof now divides into two cases of i even and i odd.

First suppose that i is even, since this is the simpler of the two cases. Since i is even, $k_{i,s_i} > 0$ and so $i + 1 = 2^{k_{i,1}} + 2^{k_{i,2}} + \cdots + 2^{k_{i,s_i}} + 1$ leading to $s_{i+1} = s_i + 1$, $k_{i+1,1} = k_{i,1}, \ldots, k_{i+1,s_i} = k_{i,s_i}$ and $k_{i+1,s_{i+1}} = 0$. So

$$K_{i+1,1} = K_{i,1}, \ldots, K_{i+1,s_i} = K_{i,s_i} \text{ and } K_{i+1,s_{i+1}} = K_{i,s_i} + 2^t = 2^t(i + 1), \tag{10.20}$$

where we use (10.19) to note that $K_{i,s_i} = 2^t \cdot i$.

By the induction hypothesis, at the end of the i-th iteration, the value of top is $s_i - 1$ and stack$[j]$ = unreducedBRW$(\tau; M_{K_{i,j}+1}, \ldots, M_{K_{i,j+1}})$ for $j \in \{0, \ldots, s_i - 1\}$. In the $(i+1)$-st iteration, Step 10 sets tmp to unreducedBRW$(\tau; M_{2^t \cdot i+1}, \ldots, M_{2^t(i+1)-1})$, i.e. to

$$\text{unreducedBRW}(\tau; M_{K_{i,s_i}+1}, \ldots, M_{K_{i+1,s_{i+1}}-1}).$$

Since $i + 1$ is odd, ntz$(i + 1) = 0$, so Step 11 sets k to 0 and as a result, the loop in Steps 12 to 14 is not executed. Step 15 updates tmp to

$$\begin{aligned} &\text{unreducedMult}(\text{reduce}(\text{unreducedBRW}(\tau; M_{K_{i,s_i}+1}, \ldots, M_{K_{i+1,s_{i+1}}-1})), M_{2^t(i+1)} + \text{keyPow}[t]) \\ =\;& \text{unreducedMult}(\text{reduce}(\text{unreducedBRW}(\tau; M_{K_{i,s_i}+1}, \ldots, M_{K_{i+1,s_{i+1}}-1})), M_{2^t(i+1)} + \tau^{2^t}). \end{aligned} \tag{10.21}$$

Step 16 increments top to $s_i = s_{i+1} - 1$ and sets stack$[s_{i+1} - 1]$ to the value in (10.21). From (10.19), $K_{i,s_i} = 2^t \cdot i$ and from (10.20), $K_{i+1,s_{i+1}} = 2^t(i + 1)$. So the expression given by (10.21) can be written as

$$\text{unreducedMult}(\text{reduce}(\text{unreducedBRW}(\tau; M_{2^t \cdot i+1}, \ldots, M_{2^t(i+1)-1})), M_{2^t(i+1)} + \tau^{2^t}). \tag{10.22}$$

The number of blocks involved in (10.22) is 2^t and so using the definition of unreducedBRW, we have

$$\begin{aligned}\mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{unreducedBRW}(\tau; M_{2^t\cdot i+1}, \ldots, M_{2^t(i+1)-1})), M_{2^t(i+1)} + \tau^{2^t}) \\ = \mathsf{unreducedBRW}(\tau; M_{2^t\cdot i+1}, \ldots, M_{2^t(i+1)}) \\ = \mathsf{unreducedBRW}(\tau; M_{K_{i+1,s_{i+1}-1}+1}, \ldots, M_{K_{i+1,s_{i+1}}}).\end{aligned} \tag{10.23}$$

To see (10.23), we note that $s_i = s_{i+1} - 1$ and so $K_{i+1,s_{i+1}-1} = K_{i+1,s_i} = K_{i,s_i} = 2^t \cdot i$ and $K_{i+1,s_{i+1}} = 2^t(i+1)$. Putting together (10.21), (10.22) and (10.23), we see that $\mathsf{stack}[s_{i+1}-1]$ contains the value as stated in the result. So at the end of the $(i+1)$-st iteration, the value of top and the entries of stack are as stated in the result.

Now suppose that i is odd. This case is more complicated since in this case $i+1$ is even and in the $(i+1)$-st iteration, the value of k in Step 11 will be positive and the loop in Steps 12 to 14 will be executed. Since i is odd, we have $k_{i,s_i} = 0$. Let $\beta \geq 1$ be such that $k_{i,s_i} = 0, k_{i,s_i-1} = 1, \ldots, k_{i,s_i-\beta+1} = \beta - 1$ and $k_{i,s_i-\beta} > \beta$. Then we can write

$$i = 2^0 + 2^1 + \cdots + 2^{\beta-1} + 2^{k_{i,s_i-\beta}} + \cdots + 2^{k_{i,1}} \quad \text{and} \quad i+1 = 2^\beta + 2^{k_{i,s_i-\beta}} + \cdots + 2^{k_{i,1}},$$

Consequently, $s_{i+1} = s_i - \beta + 1$, $k_{i+1,1} = k_{i,1}, \ldots, k_{i+1,s_i-\beta} = k_{i,s_i-\beta}$ and $k_{i+1,s_{i+1}} = k_{i+1,s_i-\beta+1} = \beta$. So

$$K_{i+1,0} = K_{i,0} = 0, \ldots, K_{i+1,s_i-\beta} = K_{i,s_i-\beta} \text{ and } K_{i+1,s_{i+1}} = K_{i+1,s_i-\beta} + 2^{t+\beta}.$$

By the induction hypothesis, at the end of the i-th iteration, $\mathsf{top} = s_i - 1$ and for $\jmath \in \{0, \ldots, s_i - 1\}$, $\mathsf{stack}[\jmath] = \mathsf{unreducedBRW}(\tau; M_{K_{i,\jmath}+1}, \ldots, M_{K_{i,\jmath+1}})$. For $\jmath = s_i - \beta, \ldots, s_i - 1$, let

$$X_\jmath = \mathsf{unreducedBRW}(\tau; M_{K_{i,\jmath}+1}, \ldots, M_{K_{i,\jmath+1}}).$$

Note that for $\imath = 0, \ldots, \beta - 1$, $2^{t+\imath}$ blocks are used in the computation of $X_{s_i-\imath-1}$.

Let us now consider what happens in the $(i+1)$-st iteration. In Step 10, tmp is assigned the value $Y = \mathsf{unreducedBRW}(\tau; M_{2^t\cdot i+1}, \ldots, M_{2^t(i+1)-1})$. Note that the computation of Y involves $2^t - 1$ blocks. From the expression for $i+1$ given above and the condition that $k_{i,s_i-\beta} > \beta$, we have $\mathsf{ntz}(i+1) = \beta$. So Step 11 sets k to β. The loop in Steps 12 to 14 is executed β times which removes β elements $X_{s_i-\beta}, \ldots, X_{s_i-1}$ from the top of the stack and adds these to the value of tmp. At the end of this loop, the value of top is $s_i - 1 - \beta$ and this value is incremented in Step 16 so that at the end of the $(i+1)$-st iteration, the value of top is $s_i - \beta = s_{i+1} - 1$ as required. For $\jmath \in \{0, \ldots, s_i - \beta - 1\}$, the value of $\mathsf{stack}[\jmath]$ does not change, and in Step 16, the new value of $\mathsf{stack}[s_i - \beta]$ is set. Our proof will be complete if we can argue that the values in stack at the end of the $(i+1)$-st iteration are as stated in the result. For $\jmath = 0, \ldots, s_i - \beta - 1$, the value in $\mathsf{stack}[\jmath]$ at the end of the $(i+1)$-st round is the same as that at the end of the i-th round and this value is $\mathsf{unreducedBRW}(\tau; M_{K_{i,\jmath}+1}, \ldots, M_{K_{i,\jmath+1}})$. As argued above, $K_{i+1,\jmath} = K_{i,\jmath}$ for $\jmath = 0, \ldots, s_i - \beta$. So the values in $\mathsf{stack}[\jmath]$, $\jmath = 0, \ldots, s_i - \beta - 1$, at the end of the $(i+1)$-st round are as stated in the result. Noting that $s_{i+1} - 1 = s_i - \beta$, the value in $\mathsf{stack}[s_{i+1} - 1]$ at the end of the $(i+1)$-st round is

$$\begin{aligned}\mathsf{unreducedMult}(\mathsf{reduce}(X_{s_i-\beta} + \cdots + X_{s_i-1} + Y), (M_{2^t(i+1)} + \mathsf{keyPow}[t+\beta])) \\ = \mathsf{unreducedMult}(\mathsf{reduce}(X_{s_i-\beta} + \cdots + X_{s_i-1} + Y), (M_{2^t(i+1)} + \tau^{2^{t+\beta}})).\end{aligned} \tag{10.24}$$

The $X_\jmath$'s are the outputs of unreduced BRW computations on consecutive blocks. Further, as mentioned above, $2^{t+\beta-1}$ blocks are used in the computation of $X_{s_i-\beta}$; $2^{t+\beta-2}$ blocks are used in the computation of $X_{s_i-\beta+1}$; and so on, finally 2^t blocks are used in the computation of X_{s_i-1}. The quantity Y is the output of an unreduced BRW computation on 2^t-1 consecutive blocks immediately following the blocks used in the computation of the $X_\imath$'s. So we have

$$\begin{aligned}
&X_{s_i-\beta}+\cdots+X_{s_i-1}+Y\\
&\quad=\ \mathsf{unreducedBRW}(\tau; M_{K_{i,s_i-\beta}+1},\ldots,M_{K_{i,s_i-\beta+1}})\\
&\qquad+\mathsf{unreducedBRW}(\tau; M_{K_{i,s_i-\beta+1}+1},\ldots,M_{K_{i,s_i-\beta+2}})\\
&\qquad+\cdots\\
&\qquad+\mathsf{unreducedBRW}(\tau; M_{K_{i,s_i-1}+1},\ldots,M_{K_{i,s_i}})\\
&\qquad+\mathsf{unreducedBRW}(\tau; M_{K_{i,s_i}+1},\ldots,M_{2^t(i+1)-1})\\
&\quad=\ \mathsf{unreducedBRW}(\tau; M_{K_{i,s_i-\beta}+1},\ldots,M_{2^t(i+1)-1})\\
&\quad=\ \mathsf{unreducedBRW}(\tau; M_{K_{i+1,s_{i+1}-1}+1},\ldots,M_{K_{i+1,s_{i+1}}-1}). \qquad (10.25)
\end{aligned}$$

The last but one equality follows from Lemma 6 and the last equality follows from $K_{i+1,s_{i+1}-1} = K_{i+1,s_i-\beta} = K_{i,s_i-\beta}$ and $K_{i+1,s_{i+1}} = 2^t(i+1)$. Using (10.25) in (10.24), we obtain

$$\begin{aligned}
&\mathsf{unreducedMult}(\mathsf{reduce}(X_{s_i-\beta}+\cdots+X_{s_i-1}+Y),(M_{2^t(i+1)}+\mathsf{keyPow}[t+\beta]))\\
&\qquad\qquad=\ \mathsf{unreducedBRW}(\tau; M_{K_{i+1,s_{i+1}-1}+1},\ldots,M_{K_{i+1,s_{i+1}}}).
\end{aligned}$$

This completes the proof. □

Proof of Theorem 4. To show that Algorithm 8 correctly computes $\mathsf{BRW}(\tau; M_1,\ldots,M_l)$, it is sufficient to show that the value of tmp in Step 24 is equal to $\mathsf{unreducedBRW}(\tau; M_1,\ldots,M_l)$. The argument is similar to the proof of Theorem 1 of [59] with modifications required to handle the different manner in which partial results are stored.

If $l < 2^t$, then the loop in Steps 9 to 23 is not executed and in Step 19, tmp is assigned the value $\mathsf{unreducedBRW}(\tau; M_1,\ldots,M_l)$ which shows the result for $l < 2^t$. So suppose $l \geq 2^t$ and let

$$\lfloor l/2^t \rfloor = 2^{k_1}+\cdots+2^{k_s},$$

$K_0 = 0$, $K_1 = 2^{t+k_1}$, $K_2 = 2^{t+k_1}+2^{t+k_2},\ldots,K_s = 2^{t+k_1}+\cdots+2^{t+k_s}$. Let $r = l \bmod 2^t$ and write $l = 2^t(2^{k_1}+\cdots+2^{k_s})+r$ so that $K_s = l-r$. From Lemma 6 we have

$$\begin{aligned}
&\mathsf{unreducedBRW}(\tau; M_1,\ldots,M_l)\\
&\quad=\ \mathsf{unreducedBRW}(\tau; M_{K_0+1},\ldots,M_{K_1})+\cdots+\mathsf{unreducedBRW}(\tau; M_{K_{s-1}+1},\ldots,M_{K_s})\\
&\qquad+\mathsf{unreducedBRW}(\tau; M_{K_s+1},\ldots,M_l)\\
&\quad=\ \mathsf{unreducedBRW}(\tau; M_{K_0+1},\ldots,M_{K_1})+\cdots+\mathsf{unreducedBRW}(\tau; M_{K_{s-1}+1},\ldots,M_{K_s})\\
&\qquad+\mathsf{unreducedBRW}(\tau; M_{l-r+1},\ldots,M_l). \qquad (10.26)
\end{aligned}$$

From Lemma 7, at the end of loop from Steps 9 to 18, $\mathsf{top} = s-1$ and for $\jmath = 0,\ldots,s-1$, $\mathsf{stack}[\jmath] = \mathsf{unreducedBRW}(\tau; M_{K_\jmath+1},\ldots,M_{K_{\jmath+1}})$. In Step 19 $\mathsf{unreducedBRW}(\tau; M_{l-r+1},\ldots,M_l)$ is assigned to tmp. The value of i in Step 20 is set to $\mathsf{wt}(\lfloor l/2^t \rfloor) = s$. The loop from Steps 21

to 23 adds the values of $\mathsf{stack}[j]$, $j = 0, \ldots, s-1$, to tmp. So in Step 24, the value of tmp is given by (10.26). This completes the proof of correctness.

The argument for the operation counts are exactly the same as in the proof of Theorem 2 of [59].

From Lemma 7, the maximum value of top is

$$\max_{1 \le i \le \lfloor l/2^t \rfloor} s_i - 1 \quad = \quad \max_{1 \le i \le \lfloor l/2^t \rfloor} \mathsf{wt}(i) - 1. \tag{10.27}$$

Claim: For $l \ge 2^t$,

$$\max_{1 \le i \le \lfloor l/2^t \rfloor} \mathsf{wt}(i) \quad \le \quad \lfloor \lg l \rfloor - t + 1. \tag{10.28}$$

Proof of claim: Note that the maximum on the left hand side of (10.28) is achieved for the value of i which is maximum in the given range and is of the form one less than some power of two. For $l = 2^t$, it can be easily verified that equality holds in (10.28). For $l = 2^{t_1}$ for some $t_1 > t$, the left hand side of (10.28) equals $\mathsf{wt}(2^{t_1 - t} - 1) = t_1 - t$, while the right hand side of (10.28) equals $t_1 - t + 1$. For $t_1 \ge t$ and $2^{t_1} < l < 2^{t_1+1}$, the right hand side of (10.27) equals $t_1 - t + 1$. Also, we have $2^{t_1 - t} \le \lfloor l/2^t \rfloor \le 2^{t_1 - t + 1} - 1$ and so the left hand side of (10.28) is at most $t_1 - t + 1$. □

Using (10.28) in (10.27), we obtain the maximum value of top to be

$$\max_{1 \le i \le \lfloor l/2^t \rfloor} \mathsf{wt}(i) - 1 \quad \le \quad \lfloor \lg l \rfloor - t. \tag{10.29}$$

So the maximum size of stack at any point in the algorithm is at most $\lfloor \lg l \rfloor - t + 1$. (Note that (10.29) is tight, for example one may choose $l = 255$ and $t = 3$; On the other hand, for $l = 256$ and $t = 3$, the bound is loose, so one cannot replace the inequality by equality.) □

10.4.4 Rationale for t-BRWHash and d-2LHash

Step 19 of Algorithm 8 performs an unreduced BRW computation on the last r blocks of the message, where $r = l \bmod 2^t$. Let us consider the implementation of Step 19. Since the value of r depends on the number l of blocks in the input, the implementation of Step 19 needs to account for all the $2^t - 1$ possible positive values of r. For each such value of r, a straight line code is required to implement the unreduced BRW computation. So the implementation of Step 19 requires a total of $2^t - 1$ separate fragments of straight line codes to implement unreduced BRW computation for all the $2^t - 1$ possible positive values of r. If $t = 2$ or 3, then this accounts for 3 or 7 fragments of straight line codes respectively, which is reasonable. On the other hand, for $t = 4$ or 5, the number of straight line code fragments is 15 or 31 respectively. Having so many fragments of straight line codes make the overall program messy, difficult to optimise and increases the code size.

The design of t-BRWHash solves the above issue. Recall that in this design, BRW is used to process a number of blocks which is a multiple of 2^t. The output of this BRW computation, the leftover blocks and the length block are then processed using Poly. So when ComputeBRW is used to compute the BRW part of t-BRWHash, the computation of

Step 19 is not required (i.e. since $r = 0$, it becomes trivial). As a result, the entire issue of using $2^t - 1$ separate fragments of straight line codes become irrelevant. This leads to a much shorter and more compact code.

We have implemented both BRWHash and t-BRWHash. For BRWHash, ComputeBRW was implemented using $t = 2$ and $t = 3$, while t-BRWHash was implemented for $t = 2, 3, 4$ and 5.

We next consider the design rationale for d-2LHash. In d-2LHash, each of the individual BRW computations is performed on δ blocks, where $\delta = 2^d - 1$ is a fixed number. For small d, the δ-block BRW computation can be performed using a straight line code. As a result, it is not required to implement Algorithm 8 at all. So the design motivation for d-2LHash is to completely avoid the implementation of Algorithm 8. This reduces the implementation complexity of the BRW part. In our implementations, we have considered $d = 2, 3, 4$ and 5. Examples of straight line code for the corresponding δ-block BRW computations are given in 10.4.2.

One may also consider values of $d \geq 6$. A problem with such values of d is that the straight line code for δ-block BRW computation becomes large. For example, if $d = 6$, then a straight line code for a 63-block BRW computation is required. With larger values of d, the storage requirement for pre-computed keys will increase and efficiency benefits will be observed for longer messages. Also, having too large a value of d may have the effect that intermediate results no longer fit in the cache, which would lead to a slowdown. Due to these reasons, we did not investigate values of $d \geq 6$.

10.4.5 Explanation of parameters

We distinguish between two types of parameters, namely design parameter and implementation parameter. The output of a hash function does not depend on an implementation parameter, i.e. if the value of an implementation parameter is changed, then for the same input, the output does not change. On the other hand, the output of a hash function does depend on a design parameter. For the same input, if the value of a design parameter is changed, then the corresponding output will be different.

The parameter g which determines the group size in Horner evaluation is an implementation parameter. Similarly, t in Algorithm 8 is also an implementation parameter. On the other hand, the parameter t in t-BRWHash and the parameter d in d-2LHash are design parameters. Also, d is a design parameter in d-Hash.

In d-2LHash, the parameter d indicates that the hash function uses straight line BRW computation on $\delta = 2^d - 1$ blocks. In d-Hash, the parameter d indicates that for messages with less than 2^d blocks polyHash is used, and for messages with at least 2^d blocks d-2LHash is used. So in d-Hash, apart from its role in d-2LHash, the parameter d also indicates the switchover point from polyHash to d-2LHash.

10.4.6 Operation counts

The four hash families are built out of various combinations of Poly and BRW. Having determined the number of integer multiplications and reductions required by Poly and BRW,

we can now specify these numbers for the hash families. For Poly, we consider both the options $g = 1$ and $g > 1$. Table 10.4 provides the operation counts for the four hash functions for $g = 1$, while Table 10.5 provides the operation counts for $g > 1$. In these tables, the columns labeled 'mult' and 'red' provide the numbers of integer multiplications and reductions required for the evaluation. The column labeled 'storage' provides the number of powers of the key τ that need to be stored, while the column labeled 'pre-comp' provides the numbers of operations that are required to compute the key powers.

Elements are stored as m-bit quantities. An integer multiplication of two m-bit quantities results in a $2m$-bit quantity, while the integer addition of two m-bit quantities results in an $(m + 1)$-bit quantity. As explained earlier, in the algorithms to compute Poly and BRW, we do not immediately reduce the result of an integer multiplication. Any subsequent additions are also performed without reduction. A reduction is performed only when the intermediate quantity is to be multiplied with another m-bit element. The reductions counted in Tables 10.4 and 10.5 are such reductions.

Remark 17. *The storage requirements for key powers in Table 10.5 are overestimates. The values are simply the sums of the number of key powers required for* BRW *and the number of key powers required for* Poly *with $g > 1$. There will typically be an overlap between these key powers, which will reduce the storage requirement. For example, suppose in d-*2LHash*, we choose $d = 5$ and $g = 8$. According to Table 10.5, the number of key powers that need to be stored is $d + 2g = 21$. Let us consider the required key powers in more details. The* BRW *computation will require the key powers $\tau, \tau^2, \tau^4, \tau^8, \tau^{16}$. The computation of $V(x)$ with $g = 8$ will require the key powers $\gamma, \gamma^2, \gamma^3, \ldots, \gamma^8$, where $\gamma = \tau^{32}$. The final computation of* Poly *will require the key powers $\tau, \tau^2, \tau^3, \ldots, \tau^8$. Out of these, τ, τ^2, τ^4 and τ^8 are also required for the* BRW *computation. So the total number of key powers that will be required to be stored is $5 + 8 + 4 = 17$. Similarly, the numbers of operations required to compute the key powers are also overestimates. Since it is messy to obtain a general formula for the exact number of key powers that are required, we have chosen to provide the simpler overestimates. Later when we consider specific values of the parameters, we provide the corresponding accurate number of required key powers.*

The computation of the key powers $\tau^2, \tau^4, \tau^8, \ldots$ required for BRW computation can be done using squarings rather than multiplications. Squarings are faster than multiplications. The operation counts in Tables 10.4 and 10.5 do not make the distinction between multiplications and squarings. In our implementations, however, we have used squarings to compute the above mentioned key powers. The computations of these key powers are the only part of the entire computation which require squarings.

From Tables 10.4 and 10.5, it may be noted that the number of multiplications required by Poly is ℓ which is the maximum among the four hash functions. BRWHash, t-BRWHash and d-2LHash require about $2+\ell/2$, $2+\ell/2+(\ell \bmod 2^t)/2$ and $1+(2^{d-1}/(2^d-1))\ell$ multiplications respectively. The number of reductions depends on the value of g and the comparison between the hash functions on this feature is more complicated. Nonetheless, from the operation counts one would expect BRWHash to be significantly faster than Poly. Our timing results reported later show that this is indeed true for $g = 1$. On the other hand, for $g > 1$ the speed improvement is much more modest. See Remark 16 for an explanation of this observation.

Table 10.4: Operation counts for the hash functions for ℓ blocks with $g = 1$. In the table $\mathfrak{m} = \ell - (\ell \bmod 2^t)$, $\delta = 2^d - 1$, and $\mathfrak{n} = \lfloor \ell/\delta \rfloor$.

				pre-comp	
	mult	red	storage	mult	red
polyHash	ℓ	ℓ	1	-	-
BRWHash	$2 + \lfloor \ell/2 \rfloor$	$2 + \lfloor \ell/4 \rfloor$	$\lfloor \lg \ell \rfloor$	$\lfloor \lg \ell \rfloor$	$\lfloor \lg \ell \rfloor$
t-BRWHash	$\ell - \lceil \mathfrak{m}/2 \rceil + 2$	$\lfloor \mathfrak{m}/4 \rfloor + \ell - \mathfrak{m} + 3$	$\lfloor \lg \mathfrak{m} \rfloor$	$\lfloor \lg \mathfrak{m} \rfloor$	$\lfloor \lg \mathfrak{m} \rfloor$
d-2LHash	$\ell + 1 - \mathfrak{n}(2^{d-1} - 1)$	$\ell + 1 - \mathfrak{n}(3 \cdot 2^{d-2} - 2)$	$d + 1$	d	d

Table 10.5: Operation counts for the hash functions for ℓ blocks with $g > 1$. In the table $\mathfrak{m} = \ell - (\ell \bmod 2^t)$, $\delta = 2^d - 1$, and $\mathfrak{n} = \lfloor \ell/\delta \rfloor$.

				pre-comp	
	mult	red	storage	mult	red
polyHash	ℓ	$\lceil \ell/g \rceil$	g	$g - 1$	$g - 1$
t-BRWHash	$\ell - \lceil \mathfrak{m}/2 \rceil + 2$	$\lfloor \mathfrak{m}/4 \rfloor + \lceil (\ell - \mathfrak{m} + 2)/g \rceil + 1$	$\lfloor \lg \mathfrak{m} \rfloor + g$	$\lfloor \lg \mathfrak{m} \rfloor + g - 1$	$\lfloor \lg \mathfrak{m} \rfloor + g - 1$
d-2LHash	$\ell + 1 - \mathfrak{n}(2^{d-1} - 1)$	$\mathfrak{n} \cdot 2^{d-2} + \lceil \mathfrak{n}/g \rceil$ $+ \lceil (\ell - \delta\mathfrak{n} + 2)/g \rceil + 1$	$d + 2g$	$d + 2g - 1$	$d + 2g - 1$

10.5 Implementation details

The implementations of the hash functions require the implementation of three arithmetic operations, namely integer addition, integer multiplication and modular reduction. Of these, the latter two operations are more complicated and require substantially more time than integer addition. So we focus on the description of integer multiplication and modular reduction. As explained in Section 10.4, for speed improvement we adopted the lazy reduction strategy. This introduces certain complications, which we explain below.

We have made 64-bit assembly implementations of the hash functions for the Intel Skylake and later generation processors. A 64-bit word will be called a limb. Depending on the choice of the prime p, elements of $\mathbb{F}_p$ have representations of different sizes.

Case $p = 2^{127} - 1$: In this case, a general element of $\mathbb{F}_p$ can be represented using 127 bits. So padded message blocks, key powers, and intermediate results have 2-limb representations for all the hash functions. For Poly evaluation, the multiplicand is a padded message block and the multiplier is a key power, both of which are 2-limb quantities. For BRW evaluation, the multiplicand is a sum of a padded message block and a key power, and the multiplier is a reduced partial result. Since padded message blocks are 126-bit quantities and key powers are 127-bit quantities, the result of the sum is a 128-bit quantity. A reduced partial result is also a 2-limb quantity. So for all the hash functions, both the multiplier and the multiplicand are 2-limb quantities. Consequently, the product can be stored in a 4-limb quantity. Since we adopt the lazy reduction strategy, the result is not immediately reduced. As explained in Section 10.4, the results of several multiplications are added together and a reduction is performed on the sum. The sum of the results of several multiplications may not fit in a 4-limb quantity, and we store such a sum in a 5-limb quantity. So the reduction algorithm is applied to a 5-limb quantity to reduce it to a 127-bit quantity.

Case $p = 2^{130} - 5$: In this case, a general element of $\mathbb{F}_p$ can be represented using 3 limbs,

where the two least significant bits of the third limb (i.e. the most significant limb) are information bits. In certain cases, instead of full reduction we apply partial reduction due to which the last three significant bits of the third limb are information bits. Compared to full reduction, partial reduction requires fewer assembly instructions and leads to overall efficiency improvement. The key τ is a 128-bit string while the key powers τ^i, $i \geq 2$, are general elements of $\mathbb{F}_p$. The representation of the message blocks depends on the hash function.

1. For polyHash, after padding, full message blocks are 129-bit strings where the 129-th bit is 1, while after padding, partial message blocks are 129-bit strings where the 129-th bit is 0. Since we know whether a message block is full or partial (only the last block can be partial), padded message blocks are stored as 2-limb quantities. For multiplications involving padded full blocks, we perform the required number of additions corresponding to the 129-th bit. Multiplications involve a padded message block and a key power. The padded message block is a 2-limb quantity (with the 129-th bit 1 if the block is full), and a key power is a 3-limb quantity, where the third limb has two information bits. The integer multiplication is of the type 2-limb x 3-limb, plus an additional number of 64-bit additions in case of full message blocks. The output is stored as a 5-limb quantity.

2. For the other hash functions, padded message blocks are defined to be 129-bit strings, where the 129-th bit is 0. So for these hash functions, padded message blocks are stored as 2-limb quantities. For BRWHash evaluation, the multiplier is a sum of a padded message block and a key power, and so in general is a 131-bit quantity. The multiplicand is an intermediate result which is kept partially reduced and stored as a 131-bit quantity. So a general multiplication for BRWHash evaluation is of the type 3-limb x 3-limb and the output is stored as a 5-limb quantity. For a 2-block message, $\mathsf{BRW}(\tau; M_1, M_2) = \tau \cdot M_1 + M_2$, and in this case the multiplication $\tau \cdot M_1$ is of the type 2-limb x 2-limb.

For both $p = 2^{127} - 1$ and $p = 2^{130} - 5$, the length block, i.e. the binary representation of the length of the message, is stored as a 1-limb quantity (see Remark 15), and this is multiplied with the key τ. Correspondingly, the multiplication involving the length block is of the type 1-limb x 2-limb.

Table 10.6 provides a summary of the general types of multiplications required by the various hash functions for the two primes. In the table, (2-limb x 3-limb)+ denotes a 2-limb x 3-limb type multiplication plus a number of 64-bit additions which arises due to the 129-th bit of a padded full message block being 1. Multiplication by the length block and the 2-limb x 2-limb multiplication required by BRWHash for a 2-block message for the prime $2^{130} - 5$ are not shown in the table.

10.5.1 Size increase due to lazy reduction

The strategy of lazy reduction in BRW evaluation requires accumulating a number of outputs of unreducedBRW computations. Such accumulation takes place at two places in Algorithm 8,

Table 10.6: Types of multiplications for the various hash functions.

	$2^{127}-1$	$2^{130}-5$
polyHash	2-limb x 2-limb	(2-limb x 3-limb)+
BRWHash	2-limb x 2-limb	3-limb x 3-limb
t-BRWHash	2-limb x 2-limb	3-limb x 3-limb
d-2LHash	2-limb x 2-limb	3-limb x 3-limb

namely in the **for** loop of Steps 12 to 14 and in the **for** loop of Steps 21 to 23. In both these loops, a number of elements from the stack are popped and added to the value of tmp. The result is not reduced. We have mentioned that we use a 5-limb quantity to store the result of the accumulation. We need to argue that a 5-limb quantity is sufficient for the purpose and does not cause any overflow. Suppose that $\mathfrak{k}$ is the number of elements that are popped from the stack and added to tmp. Then $\mathfrak{k}$ is at most the size of the stack. Recall from Theorem 4 that the maximum size of the stack is $\lfloor \lg \ell \rfloor - t + 1$ and so $\mathfrak{k} \leq \lfloor \lg \ell \rfloor - t + 1$. Assuming that messages are of lengths less than 2^{64} bits, i.e. $L < 2^{64}$, we have $\ell = \lceil L/n \rceil$ and so $\ell < 2^{57}$ for $n = 128$ and $\ell < 2^{58}$ for $n = 120$. Taking the smaller of these bounds (i.e. restricting messages to have less than 2^{57} blocks), we obtain $\mathfrak{k} < 58 - t$. Since $2 \leq t \leq 5$, we have $\mathfrak{k} < 56$. We need to argue that storing tmp as a 5-limb quantity is sufficient to store the result of $\mathfrak{k}$ additions, where $\mathfrak{k} < 56$. This argument is provided separately for the two primes.

1. For $p = 2^{127} - 1$, the multiplications are of the type 2-limb x 2-limb and the result is a 4-limb quantity. Consequently, the value of tmp computed in Step 10 of Algorithm 8 is a 4-limb quantity. So storing tmp as a 5-limb quantity allows the value of $\mathfrak{k}$ to be up to 64.

2. For $p = 2^{130} - 5$, the multiplications are of the type 3-limb x 3-limb, where the 3-limb quantities are at most 131-bit strings. Consequently, the value of tmp computed in Step 10 is a 262-bit string which is stored as a 5-limb quantity. So there are a total of $64 - 6 = 58$ bits in the fifth limb (i.e. the most significant limb) which are unused. So for $\mathfrak{k} < 56$, accumulating $\mathfrak{k}$ quantities does not lead to any overflow.

In view of the above, for both the primes $p = 2^{127} - 1$ and $p = 2^{130} - 5$, storing tmp as a 5-limb quantity is sufficient for all practical sized messages.

The strategy of lazy reduction for grouped evaluation of Poly with group size g requires accumulating the results of g multiplications. Each addition increases the size of the result by one bit and so accumulating the results of g multiplications, leads to an increase in size by g bits. In a manner similar to the above, it can be argued that for $g \leq 56$, using a 5-limb quantity to store the result of the accumulation does not lead to an overflow. The bound of 56 for g is well past the point where grouping leads to efficiency improvement for practical sized messages (see Section 10.4.1).

10.5.2 Integer multiplication

The Intel Skylake and later processors provide three instructions, namely `mulx`, `adox` and `adcx`, which permit the implementation of the so-called double carry chain strategy for multi-limb integer multiplication and squaring using the schoolbook method. The approach is outlined in two Intel white papers [48, 49] using specific examples. General algorithms for multi-limb double carry chain multiplication and squaring are described in [85]. We have used the algorithms in [85] to perform the multiplications in Table 10.6. Since the number of limbs is small (2 or 3), Karatsuba multplication will not be competitive with the schoolbook method.

10.5.3 Reduction

The prime $p = 2^{127} - 1$ is a Mersenne prime. Algorithm 4 of [85] provides a general method for reduction modulo a Mersenne prime. This algorithm reduces the output of the integer multiplication or squaring algorithm. Applied to $p = 2^{127} - 1$, Algorithm 4 of [85] will reduce a 4-limb quantity to a 2-limb (more precisely, a 127-bit) quantity. However as described above, for our present application, it is required to reduce a 5-limb quantity to a 2-limb one. This requires some modifications to Algorithm 4 of [85] which somewhat increases the number of instructions required. Our implementation of reduction modulo $p = 2^{127} - 1$ makes the required modifications. The modifications are straightforward and so we skip the details.

The prime $p = 2^{130} - 5$ is a so-called pseudo-Mersenne prime. Algorithm 5 of [85] provides a method of reduction modulo such a prime. Theorem 6.3 of [85] states the condition under which Algorithm 5 applies. Let $\delta = 5$ and $\alpha = 3$ so that $2^{\alpha-1} \leq \delta < 2^{\alpha}$. Consider the 3-limb representation of elements of $\mathbb{F}_p$, where the first two limbs are $\eta = 64$ bits long and the third limb is $\nu = 2$ bits long. According to Theorem 6.3 of [85], a condition for Algorithm 5 of [85] to apply is that $\alpha < \nu + 1$. This condition fails for the prime $p = 2^{130} - 5$. So Algorithm 5 of [85] cannot be applied to perform reduction modulo $p = 2^{130} - 5$. Below we describe a new method for performing this reduction. Computation modulo the prime $p = 2^{130} - 5$ underlies the computation of the well known hash function Poly1305. We have, however, not been able to locate the reduction method that is described below in the literature.

Write the 5-limb quantity A to be reduced as $A = a_0 + 2^{130}a_1$, where a_0 is a 130-bit non-negative integer and a_1 is a non-negative integer. Then $A \equiv a_0 + 5a_1 \pmod{2^{130} - 5}$. Write $5a_1 = a_1 + 4a_1$. A key observation is that $4a_1$ can be obtained easily from the 5-limb representation of A. If we set the two least significant bits of the third limb of A to 0, then the last three limbs (i.e. the three most significant limbs) of A provide $4a_1$. From $4a_1$, the value of a_1 can be obtained by a right shift of two places. So our reduction strategy is the following. Given A, obtain a_0, next obtain $4a_1$ as described and add to a_0, then obtain a_1 and add to the sum of a_0 and $4a_1$. This gives $a_0 + 5a_1$. From the above description of the size of representation of elements required for the lazy implementation strategy, at least the 8 most signficant bits of the fifth limb of A are 0 for the prime $2^{130} - 5$. So $4a_1$ is a 3-limb quantity, where at least the 8 most significant bits of the third limb of $4a_1$ are 0. The sum $B = a_0 + 5a_1$ computed as $a_0 + 4a_1 + a_1$ results in a 3-limb quantity. Now write

$B = b_0 + 2^{130} b_1$, where b_0 is a 130-bit non-negative integer and b_1 is a 1-limb non-negative integer whose at least 8 most significant bits are 0. So $5b_1$ fits in a single limb. We have $B \equiv b_0 + 5b_1 \pmod{2^{130} - 5}$. The next step is to compute $5b_1$ and add to b_0. This provides a 131-bit quantity. Our partial reduction strategy is not to reduce this any further. It is only at the end of the entire hash function computation, that a final reduction to a 130-bit quantity is performed.

10.5.4 Storage of key powers

For $2^{127} - 1$, the key τ is a 126-bit string which is stored as a 2-limb quantity. The higher powers of τ are 127-bit quantities and are also stored as 2-limb quantities. Since a 2-limb quantity requires 16 bytes to be stored, so for $2^{127} - 1$ each of the key powers is stored as a 16-byte quantity.

For $2^{130} - 5$, τ is a 128-bit string while the higher powers of τ are 130-bit quantities. To avoid converting from byte representation to limb representation, the key powers are stored as multi-limb quantities rather than multi-byte quantities. In the case of polyHash with $g = 1$, only τ is required and it is stored as a 2-limb quantity requiring 16 bytes. For all other cases, along with τ other higher key powers are required. In principle, τ can be stored as a 2-limb quantity and the higher powers of τ as 3-limb quantities. The non-uniformity, however, makes access to the key powers less efficient. So τ as well as the higher key powers are stored as 3-limb quantities. So each key power requires 24 bytes to be stored.

For polyHash, the evaluation can be done by using groups of size $g \geq 1$. For d-2LHash, the value of d determines the number of key powers required for the BRW part of the computation. There are two Poly computations in d-2LHash, and we have considered the same value of g for both of these computations. Depending upon the values of d and g, d-2LHash will require to store a number of key powers. BRWHash will require a number of key powers which depends upon the number of blocks in the message. Similarly, the number of key powers required by t-BRWHash depends on the number of blocks in the message and the value of g used to perform the Poly part of the computation.

In Tables 10.7a, 10.7b and 10.7c, we provide the storage requirements of the various hash functions for specific values of the parameters. This is provided in two ways, first as the number of field elements that are required to be stored, and second as the number of bytes that are required to be stored. For $2^{127} - 1$, each key power is stored as a 2-limb quantity, and so the number of bytes required to store all the key powers is 16 times the number of field elements. For $2^{130} - 5$, other than the case of polyHash with $g = 1$, in all other cases, each key power is stored as a 3-limb quantity, and so the number of bytes required to store all the key powers is 24 times the number of field elements.

10.5.5 Code

We have developed assembly code for the four hash functions modulo the two primes. The various options are as follows.

1. polyHash: Implementations with group size $g = 1, 4, 8, 16$ and 32.

Table 10.7: Storage requirements of the different hash functions.

		# bytes	
	# fld elts	$2^{127}-1$	$2^{130}-5$
$g=1$	1	16	16
$g=4$	4	64	96
$g=8$	8	128	192
$g=16$	16	256	384
$g=32$	32	512	768

(a) Storage requirement of key powers for the evaluation of polyHash.

		# bytes	
	# fld elts	$2^{127}-1$	$2^{130}-5$
#blks $=\ell$	$\lceil \lg \ell \rceil$	$16\lceil \lg \ell \rceil$	$24\lceil \lg \ell \rceil$
#blks $=\ell \geq 4$, $g=4$	$1+\lceil \lg \ell \rceil$	$16+16\lceil \lg \ell \rceil$	$24+24\lceil \lg \ell \rceil$

(b) Storage requirement of key powers for the evaluation of BRWHash and t-BRWHash. The first row is for BRWHash and t-BRWHash with $g=1$, while the second row is only for t-BRWHash with $g=4$.

		# bytes	
	# fld elts	$2^{127}-1$	$2^{130}-5$
$d=2$, $g=1$	3	48	72
$d=2$, $g=4$	7	112	168
$d=2$, $g=8$	14	224	336
$d=3$, $g=1$	4	64	96
$d=3$, $g=4$	8	128	192
$d=3$, $g=8$	15	240	360
$d=4$, $g=1$	5	80	120
$d=4$, $g=4$	9	144	216
$d=4$, $g=8$	16	256	384
$d=5$, $g=1$	6	96	144
$d=5$, $g=4$	10	160	240
$d=5$, $g=8$	17	272	408

(c) Storage requirement of key powers for the evaluation of d-2LHash.

2. BRWHash: Implementations with the parameter t in evalBRW taking the values 2 and 3.

3. t-BRWHash: Implementations with the parameter $t = 2, 3, 4$ and 5. The Poly part of the hash function has been implemented with group size $g = 1$ and 4.

4. d-2LHash: Implementations with the parameter $d = 2, 3, 4$ and 5. There are two Poly parts of the hash function. The same value of g has been used for both the parts. We have implemented the Poly parts with $g = 4$ and 8.

The hash function polyHash1305 coincides with Poly1305 for messages whose lengths are multiples of 8 (i.e. messages which are a byte stream). There are several public implementations of Poly1305. To the best of our knowledge, none of the implementations consider group size g to be greater than 1. Also, none of the implementations uses the double carry chain method for integer multiplication or the reduction algorithm that we have used. So on Intel Broadwell and later generation processors, our implementations provide new code for Poly1305.

The codes for our implementations of the hash functions are available from the following links.

```
https://github.com/kn-cs/polyHash
https://github.com/kn-cs/d2LHash
https://github.com/Sreyosi/EvalBRW-p1271
https://github.com/Sreyosi/EvalBRW-p1305
https://github.com/Sreyosi/t-BRWHash-p1271
https://github.com/Sreyosi/t-BRWHash-p1305
```

10.6 Trade-off between $2^{127} - 1$ and $2^{130} - 5$

Let $p_1 = 2^{127} - 1$ and $p_2 = 2^{130} - 5$.

There are several aspects of the trade-off. From the security point of view, it is required to compare the AXU bounds. Suppose messages of length at most L bits are to be hashed. The number of blocks is $\ell = \lceil L/n \rceil$, where $n = 120$ for p_1 and $n = 128$ for p_2. Using ℓ in Table 10.3, the AXU bound for polyHash1271 is about $2^{-132} \cdot 16L/15$, while the AXU bound for polyHash1305 is about $2^{-131} \cdot L$. Similarly, the AXU bounds for BRWHash1271, t-BRWHash1271, d-2LHash1271 and d-Hash1271 are all about $2^{-131} \cdot 16L/15 + 2^{-125}$, while the AXU bounds for BRWHash1305, t-BRWHash1305, d-2LHash1305 and d-Hash1305 are all about $2^{-130} \cdot L + 2^{-124}$. So in all cases, the AXU bounds for p_1 is about half the AXU bounds for p_2. This difference in the AXU bounds is negligible.

From an implementation point of view, there are two aspects to be considered, namely storage required for the key powers and the efficiency of computation. From Table 10.7, we see that other than polyHash with $g = 1$, the number of bytes required to store the key powers in the case of p_1 is two-thirds of the number of bytes required in the case of p_2.

Next we consider the efficiency of computation. From Table 10.2, message blocks in the case of p_1 are 120-bit strings and in the case of p_2 are 128-bit strings. So given a message,

the number of message blocks in the case of p_1 is about 16/15 times the number of message blocks in the case of p_2. Consequently, the number of multiplications required by any of the four hash functions to process a message in the case of p_1 is about 16/15 times the number of multiplications in the case of p_2. The fact that hashing in the case of p_1 requires more multiplications than hashing in the case of p_2 is one dimension of the efficiency trade-off.

The other dimension of the efficiency trade-off is the time required for a single field multiplication. From Table 10.6, one may note that the multiplications in the case of p_1 are all of 2-limb x 2-limb type, while in the case of p_2 these are either (2-limb x 3-limb)+ or 3-limb x 3-limb type. So the integer multiplication part of a field multiplication is faster in the case of p_1 than in the case of p_2. Further, since p_1 is a Mersenne prime, reduction modulo p_1 requires substantially fewer operations compared to reduction modulo p_2. The combined effect of faster integer multiplication and reduction is that a field multiplication modulo p_1 is substantially faster than a field multiplication modulo p_2.

So the overall efficiency trade-off is that hashing in the case of p_1 requires more field multiplications than hashing in the case of p_2, while the cost of an individual field multiplication modulo p_1 is less than that modulo p_2. The question then is whether for p_1 the faster speed of multiplication compensates the requirement of more multiplications? Our timing results reported in Table 10.10 of Section 10.7 shows that this is indeed the case by a very healthy margin.

To summarise, using p_1 instead of p_2 decreases the AXU bound by a negligible factor, and provides significant implementation advantages in terms of requiring lower storage and faster computation.

10.7 Timing results

There are two approaches to the implementation.

1. Pre-computed key powers. In this approach, the required key powers are pre-computed and stored. The computation and storage may be for a particular session, or on a more long term basis. With the pre-computed approach, the time for computing the key powers can be ignored while considering the time for hashing a message.

2. On-the-fly: In this approach, the required key powers are computed on a per message basis. Consequently, the time for generating the key powers is considered to be part of the entire time for hashing a message.

Depending on the application at hand, one of the above two approaches will be desirable. To understand the trade-off between the two approaches, in our timing experiments, we have obtained time measurements for both the approaches.

The timing measurements were taken on a single core of an Intel Core i5-8250U Kaby Lake processor running at 1.60GHz. During the experiments, turbo boost and hyperthreading options were turned off. The OS was Ubuntu 20.05.5 LTS and the code was compiled using gcc version 10.3.0. The following flags were used during compilation.

-march=native -mtune=native -m64 -O3 -funroll-loops -fomit-frame-pointer

The various options for the implementation of the hash functions have been described in Section 10.5.5. We have taken measurements for all the options of the four hash functions for both the primes by varying the number of message blocks from 1 to 512. This resulted in a large number of measurements. Providing all of these measurements in table form will require too much space. On the other hand, plotting so many points on a graph will make it very difficult to understand the comparative performances of the different hash functions. So we have instead adopted the following approach.

From the large set of experimental results that we have recorded, we observed the following for messages with 32 or more blocks.

1. t-BRWHash with $t = 2$ or $t = 3$ does not perform better than either $t = 4$ or $t = 5$.
2. d-2LHash with $d = 2$ or $d = 3$ does not perform better than either $d = 4$ or $d = 5$.
3. BRWHash does not perform better than either d-2LHash or t-BRWHash, for $d = 4, 5$ and $t = 4, 5$.

In view of these observations, for 32 or more blocks, we do not provide the timings for BRWHash, t-BRWHash with $t = 2$ or $t = 3$ and d-2LHash with $d = 2$ or $d = 3$. For the other hash functions, the timing results are shown in Tables 10.8, 10.9 and 10.10. We note the following points regarding these tables.

1. Timing results are provided as the number of cycles per byte.
2. Each cell of all the tables has two entries. The upper entry denotes the time required when the key powers are pre-computed, while the lower entry denotes the time required when the key powers are computed on-the-fly. The only exception is the case of polyHash with $g = 1$, since in this case, other than the key itself, no other key powers are required, and so the issue of pre-computed versus on-the-fly computation of key powers does not arise.
3. Other than Table 10.10, the time measurements in all the other tables are for a particular prime and the message size is given as number of blocks. The comparison between the hash functions based on the two primes is given in Table 10.10 and in this table, the message size is given as number of bytes.

In Tables 10.8 and 10.9, we compare polyHash, d-2LHash with $d = 4, 5$, and t-BRWHash with $t = 4, 5$. For polyHash we considered $g = 1$ and $g = 8$.

There are several dimensions to the comparison between the various options.

10.7.1 Comparison between $2^{127} - 1$ and $2^{130} - 5$

The results show that for each of the hash functions, instantiation using $2^{127} - 1$ is faster than $2^{130} - 5$ in every possible scenario that we considered. Consider Table 10.10 which compares the instantiations of the hash functions for the two primes for different length messages. For polyHash with $g = 1$, the speed gain for a 10-byte message is about 10% and it increases

Table 10.8: Cycles/byte measurements for 50 to 500 blocks for the various hash functions based on the prime $2^{127} - 1$.

		# msg blks									
		50	100	150	200	250	300	350	400	450	500
polyHash1271	$g = 1$	1.30	1.28	1.28	1.28	1.27	1.27	1.27	1.27	1.27	1.27
	$g = 8$	0.66	0.64	0.65	0.63	0.63	0.63	0.63	0.62	0.62	0.62
		0.80	0.71	0.69	0.66	0.66	0.65	0.65	0.64	0.64	0.63
d-2LHash1271	$d = 4$	0.65	0.59	0.56	0.55	0.55	0.54	0.54	0.54	0.54	0.54
		0.94	0.74	0.66	0.62	0.61	0.59	0.59	0.57	0.57	0.57
d-2LHash1271	$d = 5$	0.66	0.58	0.58	0.55	0.54	0.55	0.53	0.54	0.53	0.53
		0.98	0.74	0.68	0.63	0.60	0.60	0.58	0.58	0.57	0.56
t-BRWHash1271	$t = 4$	0.73	0.65	0.62	0.61	0.61	0.62	0.60	0.59	0.59	0.61
		0.84	0.72	0.70	0.66	0.64	0.65	0.63	0.61	0.61	0.61
t-BRWHash1271	$t = 5$	0.77	0.66	0.65	0.61	0.63	0.61	0.62	0.59	0.58	0.59
		0.92	0.71	0.70	0.65	0.71	0.63	0.64	0.62	0.62	0.61

Table 10.9: Cycles/byte measurements for 50 to 500 blocks for the various hash functions based on the prime $2^{130} - 5$.

		# msg blks									
		50	100	150	200	250	300	350	400	450	500
polyHash1305	$g = 1$	1.81	1.79	1.78	1.78	1.78	1.77	1.77	1.77	1.76	1.75
	$g = 8$	0.98	0.93	0.93	0.92	0.92	0.91	0.91	0.91	0.91	0.91
		1.18	1.03	1.00	0.97	0.96	0.95	0.94	0.93	0.93	0.93
d-2LHash1305	$d = 4$	0.88	0.83	0.81	0.80	0.79	0.79	0.79	0.79	0.78	0.78
		1.33	1.05	0.96	0.91	0.89	0.87	0.85	0.84	0.83	0.83
d-2LHash1305	$d = 5$	0.87	0.82	0.80	0.78	0.78	0.77	0.77	0.77	0.77	0.76
		1.36	1.06	0.96	0.90	0.87	0.85	0.84	0.83	0.82	0.81
t-BRWHash1305	$t = 4$	0.95	0.87	0.84	0.84	0.83	0.83	0.82	0.82	0.82	0.81
		1.16	0.98	0.95	0.96	0.89	0.87	0.87	0.85	0.85	0.84
t-BRWHash1305	$t = 5$	0.95	0.88	0.85	0.85	0.83	0.82	0.82	0.89	0.82	0.81
		1.16	0.97	0.94	0.90	0.89	0.87	0.87	0.85	0.85	0.84

Table 10.10: Cycles/byte measurements for 10 to 5000 bytes for the various hash functions based on the primes $2^{127} - 1$ and $2^{130} - 5$.

	$\mathbb{F}_{2^{130}-5}$						$\mathbb{F}_{2^{127}-1}$					
	polyHash1305		d-2LHash1305		t-BRWHash1305		polyHash1271		d-2LHash1271		t-BRWHash1271	
bytes	$g = 1$	$g = 8$	$d = 4$	$d = 5$	$t = 4$	$t = 5$	$g = 1$	$g = 8$	$d = 4$	$d = 5$	$t = 4$	$t = 5$
10	5.94	5.98	12.66	12.89	9.11	9.11	5.43	5.57	10.17	10.19	7.38	7.39
		20.22	48.48	51.51	12.40	11.92		14.11	32.55	34.38	9.99	9.85
50	3.11	2.54	3.07	3.09	2.45	2.45	2.11	1.61	2.73	2.71	2.11	2.12
		5.51	10.32	10.92	4.22	3.50		3.68	7.22	7.54	2.99	2.93
100	2.42	1.61	2.03	2.03	1.51	1.51	1.62	1.03	1.70	1.72	1.34	1.34
		3.14	5.67	5.95	2.61	2.25		2.07	3.94	4.11	1.87	1.82
500	1.90	1.04	1.00	0.97	1.04	1.03	1.35	0.72	0.71	0.68	0.81	0.81
		1.35	1.72	1.75	1.45	1.30		0.92	1.16	1.18	0.99	1.03
1000	1.77	0.96	0.87	0.86	0.92	0.89	1.30	0.66	0.62	0.61	0.68	0.68
		1.12	1.23	1.25	1.15	1.11		0.76	0.84	0.85	0.79	0.78
2000	1.79	0.94	0.81	0.80	0.86	0.84	1.29	0.66	0.58	0.57	0.63	0.65
		1.02	0.99	0.99	0.99	1.02		0.71	0.69	0.69	0.70	0.69
3000	1.73	0.93	0.81	0.78	0.84	0.83	1.27	0.63	0.55	0.55	0.61	0.60
		0.98	0.93	0.91	0.94	0.91		0.66	0.62	0.63	0.67	0.65
5000	1.72	0.91	0.79	0.77	0.82	0.81	1.27	0.63	0.54	0.55	0.61	0.60
		0.95	0.87	0.85	0.89	0.87		0.65	0.59	0.60	0.63	0.65

to about 25% for a 5000-byte message. For polyHash with $g = 8$ and pre-computed key powers, the speed gain for a 10-byte message is about 7% and it increases to about 30% for a 5000-byte message. For polyHash with $g = 8$ with keys computed on-the-fly, the speed gain is about 30% for message lengths from 10 bytes to 5000 bytes. Similar substantial speed improvements are observed for all the other hash functions.

10.7.2 Comparison between polyHash for $g = 1$ and $g = 8$

Table 10.8 provides the comparison for the prime $2^{127} - 1$, while Table 10.9 provides the comparison for the prime $2^{130}-5$. For messages having 50 or more blocks, there is a significant speed improvement by considering $g = 8$ in comparison to $g = 1$. This holds irrespective of whether the key powers are pre-computed or computed on-the-fly. For example, for the prime $2^{127}-1$, the speed improvement for a 50-block message is about 50% if the key powers are pre-computed and is about 38% if the key powers are computed on-the-fly. For a 500-block message, the speed improvement for pre-computed key powers remains about the same, while the speed improvement for on-the-fly computation of key powers increases to about 50%. Similar significant speed improvement is observed for the prime $2^{130} - 5$.

For messages having up to 32 blocks, the tables show that for very short messages, if key powers are computed on-the-fly then there is a substantial loss of speed in using $g > 1$. This, however, is expected since the time to compute the key powers is taken into the measurement, but due to the small length of the message, the benefit of using the key powers cannot be obtained. On the other hand, if the key powers are pre-computed, then substantial speed improvement is observed for $g = 4$ and $g = 8$ over $g = 1$ for messages as small as having only 3 blocks.

10.7.3 Comparison between polyHash and the various BRW-based hash functions

From a theoretical standpoint, the number of integer multiplications required by polyHash is about two times that in BRWHash and the number of reductions required is about four times that in BRWHash. So one may expect BRWHash to perform about two times faster than polyHash. One may observe that compared to polyHash with $g = 1$, BRWHash with pre-computed key powers achieve about 30% to 40% speed improvement for messages having 25 or more blocks. The picture, however, changes when polyHash is computed with $g > 1$. For short messages, BRWHash no longer provides speed improvement over polyHash. See Remark 16 for an explanation of the substantial speed gain for polyHash achieved by considering $g > 1$. If we switch to d-2LHash with pre-computed key powers, then from Tables 10.8 and 10.9, we observe speed improvements of around 10% over polyHash with $g = 8$ for messages having 50 or more blocks; if the key powers are computed on-the-fly, then the speed improvements are observed for messages having 150 or more blocks. A similar, though lower, speed improvement behaviour is observed for t-BRWHash over polyHash.

The rationale for t-BRWHash has been explained in Section 10.4.4. The timings measurements show that d-2LHash is in general faster than t-BRWHash. On the other hand, from

Table 10.7c it follows that for practical sized messages, the storage requirement for d-2LHash is more than that required for t-BRWHash.

10.7.4 The hash function d-Hash

In Section 10.2.1, it was mentioned that polyHash is the fastest when the number of blocks is small, and its performance lags behind the others when the number of blocks grows. This motivated the design of the hash function d-Hash in (10.11) which applies polyHash when the number of blocks is less than 2^d and applies d-2LHash when the number of blocks is at least 2^d. The AXU bound on d-Hash is given by Theorem 3.

We put forward 4-Hash as a secure hash function which provides the speed benefit of polyHash for short messages and the speed benefit of 4-2LHash for long messages. In particular, 4-Hash applies polyHash when the number of blocks is less than 16 and applies 4-2LHash when the number of blocks is at least 16. Instantiating 4-Hash with the prime $2^{127} - 1$ gives us the hash function 4-Hash1271 which provides the fastest hashing (among all the options) for messages of all lengths.

10.7.5 Comparison between 4-Hash1271 and Poly1305

Presently, the fastest polynomial hash function over prime order fields is Poly1305. This is a widely used hash function and is part of TLS. So it is important to determine what improvement is achieved by the best construction in the present paper over Poly1305. As per our naming convention (see Section 10.2.2), the hash function polyHash1305 is identical to Poly1305 for messages whose lengths are multiples of eight.

Based on the discussion in Section 10.7.4, the most efficient hash function for messages of all sizes is 4-Hash1271. From Table 10.10, we observe that 4-Hash1271 is faster than polyHash1305 for messages of all sizes, with the speed improvement ranging from about 8.5% (for 10-byte messages) to about 40% (for 5-kilobyte messages). This makes 4-Hash1271 an attractive target for further implementation studies to confirm whether the speed improvement over Poly1305 is indeed preserved for other implementations and on other platforms.

10.8 Timing measurements for messages with few blocks

For each prime, the measurements for the different hash functions are divided into four tables, providing measurements for block sizes from 1 to 8, from 9 to 16, from 17 to 24, and from 25 to 32. The explanation of the entries in the tables are the same as that mentioned in Section 10.7.

Table 10.11: Cycles/byte measurements for 1 to 8 blocks for the various hash functions based on the prime $2^{127} - 1$.

		# msg blks							
		1	2	3	4	5	6	7	8
polyHash1271	$g = 1$	3.35	2.23	1.90	1.73	1.65	1.58	1.54	1.50
	$g = 4$	3.38	2.27	1.52	1.23	1.24	1.08	0.98	0.95
		5.65	3.03	2.13	1.77	1.65	1.44	1.30	1.23
	$g = 8$	3.38	2.28	1.55	1.24	1.07	0.99	0.90	0.89
		9.31	5.36	3.79	2.97	2.47	2.14	1.92	1.74
	$g = 16$	3.38	2.28	1.56	1.24	1.07	0.98	0.91	0.86
		19.24	10.04	6.85	5.24	4.33	3.71	3.25	2.91
	$g = 32$	3.38	2.28	1.55	1.23	1.07	0.98	0.91	0.88
		41.94	21.33	14.43	10.93	8.87	7.50	6.51	5.77
BRWHash1271	$t = 2$	4.49	2.74	1.87	2.22	1.93	1.61	1.41	1.40
		4.99	3.39	2.23	2.72	2.29	1.95	1.67	1.74
	$t = 3$	4.44	2.77	1.91	1.80	1.47	1.27	1.21	1.40
		7.60	3.07	2.10	2.19	1.81	1.55	1.37	1.70
d-2LHash1271	$d = 2$	5.48	3.15	1.90	1.62	1.45	1.27	1.22	1.14
		14.50	7.78	4.88	3.87	3.30	2.78	2.51	2.32
	$d = 3$	6.63	3.60	2.58	2.16	1.85	1.62	1.35	1.20
		20.47	10.45	7.163	5.62	4.59	3.92	3.28	2.88
	$d = 4$	6.58	3.59	2.56	2.15	1.85	1.65	1.59	1.39
		21.52	11.01	7.52	5.88	4.82	4.11	3.72	3.26
	$d = 5$	6.57	3.58	2.57	2.16	1.85	1.64	1.59	1.40
		22.75	11.62	7.94	6.20	5.07	4.32	3.88	3.40
t-BRWHash1271	$t = 2$	4.90	2.84	2.13	2.16	2.05	1.75	1.58	1.38
		9.93	5.29	3.60	3.62	3.23	2.72	2.47	2.18
	$t = 3$	4.99	2.87	2.10	1.71	1.54	1.41	1.30	1.40
		6.56	3.71	2.65	2.40	2.28	1.92	1.75	1.85
	$t = 4$	4.90	2.83	2.07	1.71	1.61	1.41	1.28	1.20
		6.50	3.71	2.69	2.37	2.29	1.91	1.73	1.75
	$t = 5$	4.99	2.88	2.11	1.77	1.59	1.51	1.33	1.24
		6.46	3.81	2.66	2.38	2.29	1.92	1.82	1.72

Table 10.12: Cycles/byte measurements for 9 to 16 blocks for the various hash functions based on the prime $2^{127} - 1$.

		# msg blks							
		9	10	11	12	13	14	15	16
polyHash1271	$g = 1$	1.48	1.45	1.44	1.42	1.41	1.40	1.39	1.38
	$g = 4$	0.95	0.91	0.86	0.84	0.86	0.84	0.81	0.80
		1.20	1.13	1.07	1.02	1.03	1.00	0.96	0.94
	$g = 8$	0.90	0.87	0.83	0.81	0.79	0.84	0.81	0.75
		1.67	1.55	1.46	1.38	1.32	1.27	1.22	1.19
	$g = 16$	0.83	0.81	0.82	0.76	0.81	0.73	0.72	0.82
		2.67	2.45	2.29	2.14	2.05	1.95	1.84	1.81
	$g = 32$	0.82	0.81	0.77	0.76	0.75	0.81	0.78	0.78
		5.18	4.73	4.35	4.03	3.78	3.57	3.37	3.21
BRWHash1271	$t = 2$	1.34	1.20	1.10	1.12	1.13	1.03	0.99	1.00
		1.61	1.49	1.34	1.77	1.30	1.32	1.15	1.25
	$t = 3$	1.33	1.20	1.09	1.14	1.08	1.01	0.98	1.02
		1.54	1.43	1.34	1.35	1.23	1.18	1.12	1.22
d-2LHash1271	$d = 2$	1.02	1.02	0.99	0.89	0.91	0.90	0.82	0.83
		2.04	1.92	1.82	1.66	1.59	1.55	1.43	1.39
	$d = 3$	1.11	1.04	1.07	1.05	0.98	0.91	0.90	0.88
		2.60	2.39	2.30	2.16	2.02	1.88	1.79	1.72
	$d = 4$	1.30	1.25	1.19	1.22	1.14	1.07	0.87	0.84
		2.95	2.72	2.51	2.41	2.27	2.13	1.85	1.75
	$d = 5$	1.29	1.21	1.17	1.15	1.10	1.06	1.09	1.02
		3.08	2.82	2.61	2.49	2.34	2.20	2.15	2.03
t-BRWHash1271	$t = 2$	1.45	1.37	1.25	1.11	1.15	1.09	1.05	1.03
		2.14	1.93	1.79	1.64	1.65	1.53	1.47	1.45
	$t = 3$	1.42	1.32	1.22	1.18	1.34	1.11	1.07	0.99
		1.81	1.65	1.54	1.46	1.48	1.39	1.34	1.28
	$t = 4$	1.26	1.19	1.12	1.08	1.07	1.03	0.99	0.98
		1.73	1.57	1.48	1.40	1.48	1.35	1.29	1.31
	$t = 5$	1.32	1.21	1.15	1.09	1.07	1.05	1.02	0.99
		1.77	1.62	1.51	1.40	1.45	1.37	1.29	1.31

Table 10.13: Cycles/byte measurements for 17 to 24 blocks for the various hash functions based on the prime $2^{127} - 1$.

		# msg blks							
		17	18	19	20	21	22	23	24
polyHash1271	$g = 1$	1.37	1.37	1.37	1.36	1.36	1.35	1.35	1.34
	$g = 4$	0.83	0.81	0.80	0.77	0.80	0.77	0.77	0.76
		0.96	0.93	0.91	0.88	0.90	0.87	0.87	0.85
	$g = 8$	0.77	0.76	0.74	0.74	0.72	0.73	0.72	0.71
		1.17	1.13	1.10	1.07	1.05	1.03	1.01	0.99
	$g = 16$	0.83	0.80	0.79	0.78	0.77	0.76	0.76	0.75
		1.77	1.69	1.63	1.59	1.53	1.50	1.46	1.42
	$g = 32$	0.82	0.69	0.77	0.67	0.66	0.67	0.66	0.66
		3.09	2.92	2.82	2.67	2.61	2.55	2.44	2.36
BRWHash1271	$t = 2$	0.99	0.98	0.89	0.92	0.90	0.94	0.87	0.89
		1.20	1.15	1.09	1.14	1.09	1.09	1.01	1.05
	$t = 3$	0.96	0.94	0.90	0.92	0.92	0.88	0.86	0.87
		1.17	1.14	1.10	1.11	1.07	1.05	0.99	1.00
d-2LHash1271	$d = 2$	0.82	0.80	0.80	0.79	0.77	0.77	0.76	0.74
		1.36	1.30	1.28	1.25	1.21	1.19	1.16	1.12
	$d = 3$	0.85	0.86	0.84	0.84	0.81	0.79	0.79	0.76
		1.64	1.60	1.55	1.51	1.45	1.40	1.38	1.32
	$d = 4$	0.82	0.80	0.81	0.80	0.79	0.81	0.79	0.78
		1.68	1.61	1.58	1.53	1.48	1.48	1.42	1.39
	$d = 5$	1.00	0.97	0.95	0.95	0.93	0.91	0.94	0.90
		1.94	1.86	1.79	1.76	1.70	1.64	1.63	1.56
t-BRWHash1271	$t = 2$	1.07	1.02	1.00	0.94	0.97	0.90	0.93	0.86
		1.44	1.39	1.33	1.26	1.28	1.26	1.22	1.16
	$t = 3$	1.03	0.98	0.96	0.97	0.94	0.92	0.91	0.86
		1.33	1.24	1.20	1.17	1.21	1.16	1.14	1.05
	$t = 4$	1.01	0.97	0.95	0.94	0.93	0.92	0.90	0.89
		1.28	1.22	1.20	1.16	1.19	1.14	1.13	1.10
	$t = 5$	1.04	1.01	0.98	0.96	0.96	0.95	0.92	0.92
		1.36	1.33	1.25	1.23	1.30	1.22	1.17	1.17

Table 10.14: Cycles/byte measurements for 25 to 32 blocks for the various hash functions based on the prime $2^{127} - 1$.

		# msg blks							
		25	26	27	28	29	30	31	32
polyHash1271	$g = 1$	1.34	1.34	1.34	1.33	1.33	1.33	1.33	1.32
	$g = 4$	0.78	0.76	0.75	0.75	0.76	0.75	0.75	0.73
		0.87	0.84	0.83	0.82	0.84	0.82	0.82	0.80
	$g = 8$	0.73	0.71	0.71	0.70	0.69	0.69	0.69	0.68
		1.00	0.97	0.97	0.94	0.93	0.92	0.90	0.89
	$g = 16$	0.74	0.74	0.76	0.72	0.78	0.72	0.77	0.78
		1.39	1.35	1.35	1.31	1.32	1.27	1.28	1.27
	$g = 32$	0.77	0.65	0.78	0.76	0.75	0.76	0.75	0.79
		2.32	2.24	2.22	2.16	2.10	2.06	2.02	1.99
BRWHash1271	$t = 2$	0.85	0.85	0.81	0.83	0.83	0.80	0.79	0.81
		1.01	1.00	0.96	0.97	0.98	0.93	0.90	0.97
	$t = 3$	0.85	0.82	0.85	0.83	0.81	0.82	0.78	0.82
		1.00	0.96	0.93	0.95	0.92	0.92	0.90	1.18
d-2LHash1271	$d = 2$	0.74	0.75	0.75	0.74	0.74	0.71	0.72	0.71
		1.11	1.10	1.08	1.07	1.05	1.02	1.01	1.00
	$d = 3$	0.78	0.76	0.75	0.72	0.73	0.71	0.71	0.72
		1.32	1.29	1.25	1.21	1.19	1.15	1.15	1.15
	$d = 4$	0.76	0.76	0.80	0.79	0.75	0.71	0.69	0.68
		1.35	1.33	1.34	1.31	1.26	1.20	1.16	1.14
	$d = 5$	0.89	0.87	0.86	0.87	0.85	0.84	0.68	0.69
		1.52	1.48	1.45	1.44	1.41	1.37	1.20	1.18
t-BRWHash1271	$t = 2$	0.88	0.85	0.89	0.82	0.88	0.82	0.82	0.80
		1.20	1.13	1.13	1.11	1.08	1.05	1.07	1.06
	$t = 3$	0.88	0.87	0.86	0.84	0.85	0.84	0.84	0.80
		1.08	1.041	1.01	1.00	1.06	1.00	1.00	0.99
	$t = 4$	0.93	0.89	0.89	0.96	0.97	0.87	0.86	0.79
		1.15	1.09	1.08	1.08	1.08	1.05	1.05	0.96
	$t = 5$	0.96	0.92	0.91	0.90	0.91	0.88	0.88	0.78
		1.19	1.15	1.14	1.12	1.13	1.10	1.13	0.96

Table 10.15: Cycles/byte measurements for 1 to 8 blocks for the various hash functions based on the prime $2^{130} - 5$.

		# msg blks							
		1	2	3	4	5	6	7	8
polyHash1305	$g = 1$	3.44	2.97	2.65	2.39	2.27	2.20	2.12	2.09
	$g = 4$	3.40	3.09	2.22	1.95	1.78	1.72	1.55	1.47
		5.78	4.41	3.44	2.75	2.45	2.29	2.04	1.88
	$g = 8$	3.40	3.09	2.23	1.93	1.69	1.54	1.40	1.37
		11.85	7.63	5.41	4.20	3.51	3.09	2.72	2.55
	$g = 16$	3.40	3.09	2.23	1.93	1.69	1.54	1.42	1.35
		25.38	14.43	9.94	7.61	6.23	5.35	4.69	4.20
	$g = 32$	3.40	3.09	2.23	1.94	1.71	1.54	1.42	1.35
		53.46	28.40	19.27	14.60	11.85	10.02	8.69	7.70
BRWHash1305	$t = 2$	5.34	3.26	2.38	2.63	2.27	1.90	1.75	1.77
		6.01	7.09	5.04	3.80	3.15	2.70	2.42	2.63
	$t = 3$	5.42	3.27	2.36	2.22	1.85	1.64	1.47	1.68
		6.00	6.69	4.46	3.86	2.99	2.49	2.13	2.84
d-2LHash1305	$d = 2$	6.18	3.53	2.30	1.95	1.73	1.68	1.57	1.48
		21.56	11.18	7.39	5.77	4.79	4.23	3.76	3.39
	$d = 3$	7.81	4.17	3.00	2.39	2.01	1.76	1.65	1.54
		28.93	14.72	10.05	7.66	6.26	5.31	4.63	4.15
	$d = 4$	7.87	4.16	2.99	2.39	2.04	1.78	1.81	1.64
		30.62	15.56	10.61	8.07	6.57	5.56	5.05	4.49
	$d = 5$	7.79	4.11	2.94	2.33	2.01	1.75	1.79	1.62
		32.28	16.35	11.12	8.47	6.87	5.83	5.27	4.69
t-BRWHash1305	$t = 2$	5.45	3.26	2.34	2.65	2.55	2.17	1.94	1.77
		7.83	4.98	3.61	4.11	3.71	3.15	2.77	2.77
	$t = 3$	4.99	2.87	2.09	1.71	1.54	1.40	1.30	1.40
		7.84	4.96	3.57	3.30	3.04	2.61	2.32	2.69
	$t = 4$	5.67	3.19	2.32	1.91	1.65	1.47	1.36	1.39
		7.79	4.96	3.63	3.28	3.03	2.61	2.32	2.37
	$t = 5$	5.79	3.30	2.41	1.99	1.69	1.52	1.40	1.46
		8.01	4.36	3.18	2.95	2.76	2.38	2.13	2.16

Table 10.16: Cycles/byte measurements for 9 to 16 blocks for the various hash functions based on the prime $2^{130} - 5$.

		# msg blks							
		9	10	11	12	13	14	15	16
polyHash1305	$g = 1$	2.04	2.02	2.00	1.98	1.96	1.95	1.93	1.92
	$g = 4$	1.43	1.43	1.35	1.31	1.29	1.30	1.25	1.23
		1.82	1.76	1.66	1.59	1.55	1.54	1.48	1.44
	$g = 8$	1.33	1.33	1.29	1.23	1.21	1.18	1.14	1.12
		2.39	2.28	2.16	2.02	1.95	1.86	1.78	1.73
	$g = 16$	1.27	1.24	1.19	1.18	1.21	1.13	1.08	1.19
		3.84	3.55	3.30	3.14	2.97	2.91	2.69	2.63
	$g = 32$	1.29	1.24	1.19	1.18	1.12	1.09	1.11	1.19
		6.93	6.35	5.87	5.43	5.11	4.80	4.58	4.37
BRWHash1305	$t = 2$	1.63	1.51	1.44	1.48	1.41	1.34	1.30	1.34
		1.83	1.74	1.68	1.69	1.64	1.58	1.54	1.55
	$t = 3$	1.85	1.44	1.48	1.46	1.32	1.26	1.24	1.27
		2.35	2.16	2.03	2.00	1.90	1.81	1.71	1.83
d-2LHash1305	$d = 2$	1.41	1.36	1.30	1.26	1.23	1.20	1.19	1.16
		3.11	2.89	2.68	2.54	2.40	2.29	2.21	2.11
	$d = 3$	1.43	1.36	1.28	1.23	1.18	1.26	1.22	1.18
		3.75	3.45	3.22	2.97	2.79	2.76	2.62	2.49
	$d = 4$	1.53	1.45	1.36	1.31	1.25	1.21	1.17	1.13
		4.06	3.72	3.43	3.20	3.00	2.84	2.67	2.55
	$d = 5$	1.51	1.44	1.36	1.31	1.24	1.20	1.26	1.21
		4.24	3.87	3.58	3.34	3.12	2.96	2.89	2.74
t-BRWHash1305	$t = 2$	1.80	1.66	1.55	1.49	1.53	1.45	1.39	1.34
		2.70	2.47	2.29	2.16	2.15	2.02	1.92	1.97
	$t = 3$	1.42	1.32	1.22	1.18	1.34	1.11	1.07	0.99
		2.62	2.40	2.24	2.10	2.09	1.98	1.89	1.88
	$t = 4$	1.35	1.25	1.20	1.15	1.11	1.07	1.03	1.24
		2.33	2.14	2.01	1.89	1.90	1.79	1.72	1.89
	$t = 5$	1.39	1.28	1.22	1.18	1.13	1.11	1.06	1.10
		2.12	1.95	1.84	1.73	1.77	1.65	1.59	1.66

Table 10.17: Cycles/byte measurements for 17 to 24 blocks for the various hash functions based on the prime $2^{130} - 5$.

		# msg blks							
		17	18	19	20	21	22	23	24
polyHash1305	$g = 1$	1.91	1.91	1.90	1.89	1.88	1.88	1.87	1.87
	$g = 4$	1.22	1.23	1.20	1.18	1.18	1.19	1.17	1.15
		1.42	1.42	1.39	1.35	1.34	1.34	1.32	1.29
	$g = 8$	1.12	1.13	1.11	1.10	1.08	1.07	1.05	1.05
		1.69	1.67	1.63	1.58	1.55	1.51	1.48	1.45
	$g = 16$	1.21	1.18	1.15	1.13	1.12	1.10	1.09	1.08
		2.56	2.46	2.37	2.29	2.22	2.15	2.10	2.04
	$g = 32$	1.07	1.15	1.05	1.05	1.76	1.11	1.06	1.14
		4.15	4.00	3.81	3.66	4.19	3.44	3.36	3.25
BRWHash1305	$t = 2$	1.31	1.24	1.22	1.25	1.22	1.20	1.18	1.18
		1.54	1.49	1.45	1.48	1.44	1.41	1.40	1.46
	$t = 3$	1.21	1.15	1.19	1.27	1.13	1.10	1.11	1.13
		2.00	1.70	1.63	1.66	1.57	1.56	1.49	1.54
d-2LHash1305	$d = 2$	1.14	1.12	1.11	1.09	1.08	1.08	1.07	1.06
		2.06	1.97	1.92	1.86	1.80	1.77	1.73	1.69
	$d = 3$	1.15	1.12	1.10	1.07	1.10	1.08	1.06	1.04
		2.38	2.28	2.20	2.11	2.10	2.03	1.97	1.92
	$d = 4$	1.10	1.08	1.05	1.03	1.01	1.05	1.03	1.01
		2.43	2.33	2.24	2.16	2.08	2.08	2.01	1.96
	$d = 5$	1.17	1.15	1.12	1.10	1.07	1.05	1.10	1.07
		2.62	2.51	2.41	2.32	2.24	2.17	2.17	2.09
t-BRWHash1305	$t = 2$	1.38	1.74	1.28	1.30	1.29	1.25	1.22	1.22
		1.98	1.89	1.82	1.77	1.78	1.71	1.66	1.62
	$t = 3$	1.03	0.98	0.96	0.97	0.94	0.92	0.91	0.86
		1.90	1.81	1.81	1.70	1.70	1.65	1.60	1.52
	$t = 4$	1.28	1.23	1.19	1.17	1.14	1.12	1.09	1.12
		1.90	1.84	1.75	1.69	1.70	1.65	1.62	1.56
	$t = 5$	1.10	1.06	1.03	1.01	0.99	0.98	0.96	0.99
		1.68	1.61	1.56	1.51	1.54	1.48	1.45	1.48

Table 10.18: Cycles/byte measurements for 25 to 32 blocks for the various hash functions based on the prime $2^{130} - 5$.

		# msg blks							
		25	26	27	28	29	30	31	32
polyHash1305	$g = 1$	1.87	1.86	1.86	1.85	1.85	1.85	1.85	1.84
	$g = 4$	1.15	1.16	1.14	1.13	1.13	1.14	1.12	1.11
		1.28	1.29	1.27	1.25	1.24	1.25	1.23	1.22
	$g = 8$	1.05	1.06	1.05	1.04	1.03	1.03	1.01	1.01
		1.44	1.44	1.41	1.38	1.37	1.35	1.33	1.31
	$g = 16$	1.08	1.08	1.08	1.05	1.10	1.10	1.10	1.11
		2.01	1.98	1.95	1.89	1.90	1.88	1.85	1.83
	$g = 32$	1.13	1.02	1.08	1.01	1.02	1.03	1.04	1.12
		3.17	3.07	2.98	2.92	2.85	2.80	2.72	2.72
BRWHash1305	$t = 2$	1.16	1.13	1.14	1.15	1.13	1.10	1.12	1.12
		1.53	1.48	1.46	1.48	1.46	1.40	1.39	1.47
	$t = 3$	1.12	1.08	1.09	1.07	1.04	1.02	1.04	1.06
		1.64	1.58	1.55	1.52	1.54	1.49	1.46	1.50
d-2LHash1305	$d = 2$	1.06	1.05	1.07	1.05	1.04	1.04	1.03	1.02
		1.67	1.64	1.63	1.60	1.57	1.55	1.53	1.50
	$d = 3$	1.02	1.01	0.99	1.01	1.02	0.99	0.98	0.97
		1.86	1.81	1.77	1.76	1.74	1.69	1.66	1.63
	$d = 4$	1.00	0.98	0.97	0.96	0.95	0.98	0.97	0.96
		1.90	1.85	1.81	1.76	1.73	1.74	1.70	1.67
	$d = 5$	1.05	1.04	1.03	1.02	1.00	0.98	0.94	0.94
		2.03	1.98	1.93	1.89	1.84	1.81	1.73	1.70
t-BRWHash1305	$t = 2$	1.23	1.19	1.17	1.15	1.19	1.16	1.14	1.13
		1.64	1.58	1.55	1.52	1.54	1.49	1.46	1.50
	$t = 3$	0.88	0.87	0.86	0.84	0.85	0.84	0.84	0.80
		1.54	1.49	1.46	1.43	1.45	1.42	1.40	1.40
	$t = 4$	1.11	1.08	1.06	1.05	1.04	1.02	1.00	1.02
		1.63	1.54	1.53	1.47	1.49	1.45	1.43	1.41
	$t = 5$	0.99	0.97	0.96	0.95	0.94	0.92	0.91	1.03
		1.45	1.40	1.37	1.34	1.38	1.33	1.31	1.44

10.9 Summary

A major finding of the present work is that for polynomial hashing over prime order fields, using the prime $2^{127} - 1$ results in significantly faster hashing compared to the prime $2^{130} - 5$. While this finding is based on our specific implementations for Intel processors, we do not envisage any platform where using $2^{127} - 1$ will result in slower hashing than using $2^{130} - 5$. Confirming (or not) this prediction is a possible direction of future implementation work.

The other major finding is that a judicious mix of Poly and BRW polynomials can lead to significant speed improvement over using only Poly.

Chapter 11

Conclusion And Future Research Possibilities

This part of the thesis studied constructions and efficient implementations of universal hash functions defined over prime order fields. Two types of polynomials have been considered-classical polynomials and BRW polynomials. Techniques considered for evaluation of the polynomials were Horner's method and evaluation process of BRW polynomials.

In Chapter 9 a simple algorithmic modification helps to apply 4-decimation Horner [32] throughout the length of the input message which in turn gives speed-up for small-length messages. We introduced a *balancing technique* which gives an improved vectorized implementation of Poly1305 [16, 60]. We have reported the improved performance for several compilers and micro-architectures.

In a recent work [40] the primes $2^{266}-3,\ 2^{174}-3,\ 2^{150}-3,\ 2^{122}-3,\ 2^{116}-3$ have been proposed to instantiate prime order fields to construct universal hash functions similar to the hash function of Poly1305 [16]. Competitive results have been presented in this work with respect to Poly1305 for sequential implementation. Studying the effect of this *balancing technique* in the SIMD implementation for polynomials over the prime order fields proposed in [40] may be a possible scope of work.

In Chapter 10 we have proposed several constructions of universal hash functions based on polynomials defined over prime order fields. We have considered the prime order fields $\mathbb{Z}_{2^{130}-5}$ and $\mathbb{Z}_{2^{127}-1}$ for this purpose. Classical polynomials and BRW polynomials were instantiated with coefficients from the said prime order fields. New universal hash functions have been proposed combining Horner's method and evaluation method of BRW polynomials.

A direction of future work is to examine whether the speed improvement holds for SIMD implementations. For Poly1305, SIMD implementations have been reported in [60, 24]. New SIMD implementation of polyHash1271 remains to be done. For BRW, parallel hardware implementation for binary extension fields have been reported in [33]. Whether these can be translated to SIMD implementation in software, or whether there is a different method of implementing BRW using SIMD in software is a topic for future research. The other direction of possible future implementation work is to obtain implementations of the hash functions proposed in Chapter 10 on ARM processors.

Chapter 12

Concluding The Thesis

Broadly speaking this thesis has contributed in both of the broad domains of Cryptology: symmetric and public-key Cryptology. In the former domain, design and implementations of several polynomial hash functions over prime order fields have been proposed. The new constructions surpass the performance of the hash function of Poly1305 in terms of *cycles per byte*. The vectorized implementation due to the *balancing technique* gives improved performance with respect to the Goll-Gueron version [60]. In the latter domain, the thesis studies generic decoding algorithms called Information Set Decoding algorithms used for decoding random linear error-correcting codes. These algorithms are useful for cryptanalysis of code-based public-key encryption schemes. The only inputs for carrying out cryptanalysis are the public parity-check matrix and a syndrome vector. In this context a generalisation of the Stern's ISD algorithm along with a novel method of obtaining time/memory trade-off points have been proposed. We have studied the performance of the generalisation of Stern's algorithm and the effective set of TMTO points for the KEM named Classic McEliece.

Possible future directions of research arising out of each of the two parts of this thesis have been discussed in Chapter 6 and Chapter 11. In this chapter some additional use cases of the primitives proposed in Chapter 10 and the generalisation discussed in Chapter 4 will be presented.

Information set decoding algorithms solve an instance of the *Syndrome Decoding Problem*. As mentioned in Chapter 3 the *Information Set Decoding* technique works best when ω is lesser than the *Gilbert-Varshamov bound*. So in such a setting it is the main tool available for cryptanalysis of a cryptosystem whose security is based on computational difficulty of decoding a random linear code. Any ISD algorithm accepts the parity-check matrix $\mathbf{H_0}$ along with the syndrome vector $\mathbf{s_0}$ and proceeds to *guess* an error vector $\mathbf{e}$ without using any additional knowledge about the structure of the underlying linear error-correcting code. In the brief survey presented in Chapter 3 we have seen that with the advancement of ISD algorithms better ideas to do the brute force search of an error vector have come up which succeed to bring down time complexity but the more advanced ISD algorithms like MMT [78], BJMM [10], new time/memory trade-off of MMT [53] etc have higher memory complexity than the simpler ones like Lee-Brickell [76], Stern [101] etc. As pointed out in [30, 3] the performance gain of advanced ISD algorithms over the simpler ones become less significant in $\mathbb{F}_q$ with $q > 2$. So cryptanalysis using advanced ISD algorithms with such inputs may demand

higher memory complexity. For example in the new signature scheme like SDiTH [30] the parity check matrix is defined over larger finite fields like $\mathbb{F}_{251}$ and $\mathbb{F}_{256}$. The performance of simpler algorithms like Lee-Brickell in $\mathbb{F}_q$ and Stern [101] ([90] gives $\mathbb{F}_q$ version of Stern's attack) will be interesting in such cases. Adapting the generalised Stern's attack [23] in $\mathbb{F}_q$ and studying its performance may be a scope of future work. Using this generalised version along with the effective time/memory trade-off for cryptanalysis of schemes like [30] may be interesting.

On the other hand, polynomial-evaluation based universal hash functions have been used to instantiate the keyed hash function of Wegman-Carter type of Message Authentication Codes as briefly presented in Chapter 8. Message Authentication codes are further used to build Authenticated Encryption with Associated Data schemes. Based on these two cryptographic schemes we discuss two use cases of the new hash functions proposed in Chapter 10.

- *Use Case 1.* The implementation in [16] by Bernstein used key clamping to leverage the FPU because this implementation [16] used floating point evaluations. The key clamping free version of the hash function of Poly1305 has been named as polyHash1305 in Chapter10. Since polyHash1305 does not use key clamping, its security level is higher than the hash function of Poly1305. We do not consider key clamping for any of our constructions. This chapter proposes several instances of universal hash functions which are at the same security level (please refer to Table 10.3 for detailed security bounds) as that of polyHash1305 in terms of Almost Universal Hash (AXU) definition or at least at a security level which gives a competitive trade-off. Section 10.6 clearly shows the trade-off between the two prime numbers. The security levels and the performances in terms of *cycles per byte* of each of these hash functions suggest that they may be used to replace the hash function used in Poly1305. The hash functions instantiated over $\mathbb{Z}_{2^{130}-5}$ are at same level of security as polyHash1305 in terms of AXU while those instantiated over $\mathbb{Z}_{2^{127}-1}$ are at bit lower security level but all of the alternative hash functions can be evaluated faster than that of polyHash1305. These encourage to use each of the new primitives to obtain new message authentication codes.

- *Use Case 2.* ChaCha20-Poly1305, a widely used AEAD scheme, uses the MAC Poly1305 to authenticate the variable length associated data along with the encrypted message output by the ChaCha20 stream cipher. Apart from TLS, this AEAD scheme is also used in OpenSSH, Lightning Network, OTRv4, WireGuard, IPSec etc. A possible use case of the hash functions proposed in Chapter10 is to use them in such AEAD schemes. Each one of the proposed hash functions performs better than polyHash1305 while maintaining the same security level. So the new MACs obtained from *Use Case 1* can be used in combination with ChaCha20 to build new AEAD schemes which can be used in a number of real life applications.

Further discussions The aims behind proposing advanced polynomial hashing includes the following notions.

- better security : Choice of the prime number heavily determines the ϵ-AXU bound.

- Faster evaluation and efficient implementations: More than one factor determines speeding up the evaluation of a polynomial. The ones explored in this thesis includes the following:
  - choice of prime number: number of limbs influences cost of field multiplications,
  - method of evaluation: Horner and BRW,
  - style of reduction.
- security/performance trade-off

The new constructions offer faster evaluation at security level of Poly1305. For higher security larger prime numbers, for instance $2^{266} - 3$, $2^{452} - 3$, $2^{444} - 17$, $2^{521} - 1$ may be considered to explore BRW-hashing, t-BRWHash and d-2LHash in larger prime order fields.

Studying single-user and multi-user security levels of AEAD schemes obtained by replacing Poly1305 with the new proposed hash functions are useful research problems. Degabriele et al. in [41] have studied the security of ChaCha20-Poly1305 in multi-user settings. They have proposed the prime number $2^{255} - 19$ as an alternate to $2^{130} - 5$. Studying the various combinations of polynomial hashing and BRW-hashing may be explored for this prime number as another future research direction.

Bibliography

[1] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. `https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf` (last accessed on July17, 2024, 2020.

[2] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process . `https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf` (last accessed on July 17, 2024.

[3] Alexander Meurer. A Coding-Theoretic Approach to Cryptanalysis. `https://www.crypto.ruhr-uni-bochum.de/imperia/md/content/diss.pdf`(accessed on 14th Jan, 2025), 2012.

[4] Nicolas Aragon, Paulo Barreto, Loïc Bidoux Slim Bettaieb, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zemor, Valentin Vasseur, Santosh Ghosh, and Jan Richter-Brokmann. BIKE. `https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/round-4/submissions/BIKE-Round4.zip`. Round 4 submission, 2022.

[5] M. Atici and Douglas R. Stinson. Universal Hashing and Multiple Authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, page 16–30, Berlin, Heidelberg, 1996. Springer-Verlag.

[6] Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. A Fast Provably Secure Cryptographic Hash Function. *IACR Cryptology ePrint Archive*, 2003:230, 01 2003.

[7] Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. A family of fast syndrome based cryptographic hash functions. In Ed Dawson and Serge Vaudenay, editors, *Progress in Cryptology – Mycrypt 2005*, pages 64–83, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[8] Roberto Avanzi, Joppe Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien

Stehle. CRYSTALS-KYBER: algorithm specifications and supporting documentation. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions. 2009.

[9] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. A Finite Regime Analysis of Information Set Decoding Algorithms. *Algorithms*, 12(10):209, 2019.

[10] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in $2^{n/20}$: How $1 + 1 = 0$ improves information set decoding. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012 Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, page 520–536. Springer.

[11] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems (Corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.

[12] Daniel J. Bernstein. ChaCha, a variant of Salsa20. Workshop Record of SASC 2008: The State of the Art of Stream Ciphers, January 2008, http://cr.yp.to/chacha/chacha-20080128.pdf.

[13] Daniel J. Bernstein. Floating-point arithmetic and message authentication. https://cr.yp.to/papers.html#hash127. 2004.

[14] Daniel J. Bernstein. Polynomial evaluation and message authentication. http://cr.yp.to/papers.html#pema. 2007.

[15] Daniel J. Bernstein. The Salsa20 family of stream ciphers. http://cr.yp.to/papers.html#salsafamily. Document ID: 31364286077dcdff8e4509f9ff3139ad. Date: 2007.12.25.

[16] Daniel J. Bernstein. The Poly1305-AES message-authentication code. In Helena Handschuh Henri Gilbert, editor, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2005.

[17] Daniel J. Bernstein. Polynomial Evaluation and Message Authentication, 2007. http://cr.yp.to/papers.html#pema.

[18] Daniel J. Bernstein and Tung Chou. CryptAttackTester: high-assurance attack analysis. Cryptology ePrint Archive, Paper 2023/940, 2023. https://eprint.iacr.org/2023/940.

[19] Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai,

Martin Tomlinson, and Wen Wang. Classic McEliece, Round 4 submission. `https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/round-4/submissions/mceliece-Round4.tar.gz`. Round 4 submission, 2022.

[20] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the McEliece Cryptosystem. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, pages 31–46, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[21] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The Security Impact of a New Cryptographic Library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer, 2012.

[22] Sreyosi Bhattacharyya, Kaushik Nath, and Palash Sarkar. Polynomial hashing over prime order fields. *Advances in Mathematics of Communications*, 2024.

[23] Sreyosi Bhattacharyya and Palash Sarkar. Concrete Time/Memory Tade-Offs in Generalised Stern's ISD Algorithm. In *Proceedings of Indocrypt 2023*, Lecture Notes in Computer Science, pages 307–328. Springer.

[24] Sreyosi Bhattacharyya and Palash Sarkar. Improved SIMD Implementation of Poly1305. In *IET Information Security*, volume 14, pages 521–530, 2020.

[25] Leif Both and Alexander May. Decoding linear codes with high error rate and its impact for LPN security. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA*, volume 10786 of *Lecture Notes in Computer Science book series*, pages 25–46. Springer, 2018.

[26] Leif Both and Alexander May. Decoding linear codes with high error rate and its impact for lpn security. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 25–46, Cham, 2018. Springer International Publishing.

[27] Rémi Bricout, André Chailloux, Thomas Debris-Alazard, and Matthieu Lequesne. Ternary Syndrome Decoding with Large Weight. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography – SAC 2019*, pages 437–466, Cham, 2020. Springer International Publishing.

[28] A. Canteaut and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511. *IEEE Transactions on Information Theory*, 44(1):367–378, 1998.

[29] Anne Canteaut and Nicolas Sendrier. Cryptanalysis of the original McEliece Cryptosystem. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology — ASIACRYPT'98*, pages 187–199, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[30] Carlos Aguilar Melchor, Thibauld Feneuil, Nicolas Gama, Shay Gueron, James Howe, David Joseph, Antoine Joux, Edoardo Persichetti, Tovohery H. Randrianarisoa, Matthieu Rivain, Dongze Yue . The Syndrome Decoding in the Head (SD-in-the-Head) Signature Scheme Algorithm Specifications and Supporting Documentation – Version 1.0 . `https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/SDitH-spec-web.pdf`.

[31] Kévin Carrier, Thomas Debris-Alazard, Charles Meyer-Hilfiger, and Jean-Pierre Tillich. Statistical Decoding 2.0: Reducing Decoding to LPN. In *Advances in Cryptology – ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part IV*, page 477–507, Berlin, Heidelberg, 2023. Springer-Verlag.

[32] Debrup Chakraborty, Sebati Ghosh, and Palash Sarkar. A Fast Single-Key Two-Level Universal Hash Function. *IACR Transactions on Symmetric Cryptology*, pages 106–128, 03 2017.

[33] Debrup Chakraborty, Cuauhtemoc Mancillas-López, Francisco Rodríguez-Henríquez, and Palash Sarkar. Efficient Hardware Implementations of BRW Polynomials and Tweakable Enciphering Schemes. *IEEE Trans. Computers*, 62(2):279–294, 2013.

[34] Debrup Chakraborty, Cuauhtemoc Mancillas-López, and Palash Sarkar. STES: A stream cipher based low cost scheme for securing stored data. *IEEE Trans. Computers*, 64(9):2691–2707, 2015.

[35] J.T. Coffey and R.M. Goodman. The complexity of information set decoding. *IEEE Transactions on Information Theory*, 36(5):1031–1037, 1990.

[36] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[37] N Courtois, M Finiasz, and N.Sendrier. How to achieve a McEliece-based digital signature scheme. In *In Advances in Cryptology - ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science book series*, pages 157–174. Springer-Verlag.

[38] Tanja Lange Daniel J. Bernstein and Christiane Peters. Smaller decoding exponents: ball-collision decoding. In *In Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science book series*, pages 743–760. Springer.

[39] Thomas Debris-Alazard, Nicolas Sendrier, and Jean-Pierre Tillich. WAVE: A New Family of Trapdoor One-Way Preimage Sampleable Functions Based on Codes. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and*

Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part I, volume 11921 of *Lecture Notes in Computer Science*, pages 21–51. Springer, 2019.

[40] J. Degabriele, J. Gilcher, J. Govinden, and K. G. Paterson. SoK: Efficient Design and Implementation of Polynomial Hash Functions over Prime Fields. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 131–131, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[41] Jean Paul Degabriele, Jérôme Govinden, Felix Günther, and Kenneth G. Paterson. The Security of ChaCha20-Poly1305 in the Multi-User Setting. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1981–2003, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Bert den Boer. A simple and key-economical unconditional authentication scheme. *Journal of Computer Security*, 2:65–71, 1993. URL: `http://cr.yp.to/bib/entries.html#1993/denboer`.

[43] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, 22(6):644–654, 1976.

[44] Whitfield Diffie and Martin E. Hellman. Special Feature Exhaustive Cryptanalysis of the NBS Data Encryption Standard. *Computer*, 10:74–84, 1977.

[45] Itai Dinur, Orr Dunkelman, Nathan Keller, and Adi Shamir. Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 719–740, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[46] Ilya Dumer. On minimum distance decoding of linear codes. In *In Proceedings of the 5th Joint Soviet-Swedish International Workshop on Information Theory, 1991*, pages 50–52.

[47] Morris J. Dworkin. SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Technical report, Gaithersburg, MD, USA, 2007.

[48] V. Gopal E. Ozturk, J. Guilford and W. Feghali. New instructions supporting large integer arithmetic on Intel architecture processors, intel white paper. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf`. 2012.

[49] J.Guilford E.Ozturk and V.Gopal. Large integer squaring on Intel architecture processors, Intel white paper. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/large-integer-squaring-ia-paper.pdf`. 2013.

[50] Andre Esser and Emanuele Bellini. Syndrome Decoding Estimator. In *Public-Key Cryptography – PKC 2022*, volume 13177 of *Lecture Notes in Computer Science book series*, pages 112–141. Springer.

[51] Andre Esser, Alexander May, and Floyd Zweydinger. McEliece Needs a Break – Solving McEliece-1284 and Quasi-Cyclic-2918 with Modern ISD. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022*, pages 433–457, Cham, 2022. Springer International Publishing.

[52] Andre Esser, Javier Verbel, Floyd Zweydinger, and Emanuele Bellini. `CryptographicEstimators`: a Software Library for Cryptographic Hardness Estimation. Cryptology ePrint Archive, Paper 2023/589, 2023. `https://eprint.iacr.org/2023/589`.

[53] Andre Esser and Floyd Zweydinger. New time-memory trade-offs for subset sum improving ISD in theory and practice. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023- 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part V*, volume 14008 of *Lecture Notes in Computer Science*, page 360–390. Springer, 2023.

[54] Thibauld Feneuil, Antoine Joux, and Matthieu Rivain. Syndrome Decoding in the Head: Shorter Signatures from Zero-Knowledge Proofs. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 541–572, Cham, 2022. Springer Nature Switzerland.

[55] Matthieu Finiasz and Nicolas Sendrier. Security Bounds for the Design of Code-Based Cryptosystems. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 88–105, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[56] Jean-Bernard Fischer and Jacques Stern. An efficient pseudo-random generator provably as secure as syndrome decoding. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 245–255, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[57] Kirk Fleming. ROUND 3 OFFICIAL COMMENT: Classic McEliece, started on november 10, 2020, 7:05:28 am. The Sage Development Team, `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/EiwxGnfQgec?pli=1(accessedon9thAugust,2023)`, 2020.

[58] Ernst Gabidulin. Theory of codes with maximum rank distance (translation). *Problems of Information Transmission*, 21:1–12, 01 1985.

[59] Sebati Ghosh and Palash Sarkar. Evaluating Bernstein–Rabin–Winograd polynomials. *Designs, Codes and Cryptography*, 87, 03 2019.

[60] Martin Goll and Shay Gueron. Vectorization of Poly1305 Message Authentication Code. In *Proceedings of the 2015 12th International Conference on Information Technology - New Generations*, ITNG '15, page 145–150, USA, 2015. IEEE Computer Society.

[61] Shay Gueron. AES-GCM-SIV implementations (128 and 256-bit). Technical report, 2016.

[62] Shay Gueron and Michael E. Kounavis. Intel carry-less multiplication instruction and its usage for computing the gcm mode. Intel white paper. Technical report, 2010.

[63] Shai Halevi and Hugo Krawczyk. MMH: Software message authentication in the Gbit/second rates. In Eli Biham, editor, *Fast Software Encryption*, pages 172–189, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[64] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *J. ACM*, 21:277–292, 04 1974.

[65] Nick Howgrave-Graham and Antoine Joux. New Generic Algorithms for Hard Knapsacks. In H. Gilbert, editor, *Annual International Conference on the Theory and Applications of Cryptographic Techniques EUROCRYPT 2010: Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science book series*. Springer.

[66] Intel. Intel® Intrinsics Guide. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#avxnewtechs=AVX2`, last accessed on July 17, 2024.

[67] Pierre Karpman and Charlotte Lefevre. Time-Memory Tradeoffs for Large-Weight Syndrome Decoding in Ternary Codes . In Yohei Watanabe Goichiro Hanaoka, Junji Shikata, editor, *25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8–11, 2022, Proceedings, Part I*, volume 13177 of *Lecture Notes in Computer Science book series*. Public-Key Cryptography – PKC 2022.

[68] Donald E. Knuth. Art of Computer Programming, Volume 4A, The Combinatorial Algo- rithms, Part1.

[69] K. Kobara and H. Imai. Semantically secure McEliece public-key cryptosystems - conversions for McEliece PKC. In *In Practice and Theory in Public Key Cryptography - PKC 2001 Proceedings*. Springer-Verlag.

[70] Tadayoshi Kohno, John Viega, and Doug Whiting. CWC: A High-Performance Conventional Authenticated Encryption Mode. In Bimal Roy and Willi Meier, editors, *Fast Software Encryption*, pages 408–426, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[71] Hugo Krawczyk. LFSR-based Hashing and Authentication. In *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '94, page 129–139, Berlin, Heidelberg, 1994. Springer-Verlag.

[72] T. Krovetz. RFC 4418: UMAC: Message Authentication Code using Universal Hashing, 2006.

[73] Ted Krovetz and Phillip Rogaway. Fast Universal Hashing with Small Keys and No Preprocessing: The PolyR Construction. In Dongho Won, editor, *Information Security and Cryptology - ICISC 2000, Third International Conference, Seoul, Korea, December 8-9, 2000, Proceedings*, volume 2015 of *Lecture Notes in Computer Science*, pages 73–89. Springer, 2000.

[74] Mark N. Wegman Larry Carter. Universal classes of hash functions. In *J. Comput. Syst. Sci.*, volume 18, pages 143–154, 1979.

[75] Mark N. Wegman Larry Carter. New hash functions and their use in authentication and set equality. In *J. Comput. Syst. Sci.*, volume 22, pages 265–279, 1981.

[76] Pil Joong Lee and Ernest F. Brickell. An observation on the security of McEliece's public-key cryptosystem. In Christoph G. Gunther, editor, *Advances in Cryptology - EUROCRYPT '88, Workshop on the Theory and Application of of Cryptographic Techniques, Davos, Switzer- land*, volume 330 of *Lecture Notes in Computer Science book series*, pages 275–280. Springer, 1988.

[77] Jeffrey S. Leon. A probabilistic algorithm for computing minimum weights of large error- correcting codes. *IEEE Trans. Inf. Theory*, 34(5):1354–1359, 1988.

[78] Alexander May, Alexander Meurer, and Enrico Thomae. Decoding random linear codes in $\tilde{O}(2^{0.054n})$. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security*, volume 7073 of *Lecture Notes in Computer Science*. Springer, 2011.

[79] Alexander May and Ilya Ozerov. On computing nearest neighbors with applications to decoding of binary linear codes. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria*, volume 9056 of *Lecture Notes in Computer Science book series*, pages 203–228. Springer, 2015.

[80] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. `http://ipnpr.jpl.nasa.gov/progressreport2/42-44/44N.PDF`. Propulsion Laboratory DSN Progress Report 42–44, 1978.

[81] Carlos Aguilar Melchor, Nicolas Aragon, Magali Bardet, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Ayoub Otmani, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. ROLLO. `https://pqc-rollo.org/`.

[82] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Maxime Bros, Alain Couvreur, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, and Gilles Zémor. RQC. https://pqc-rqc.org/.

[83] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, and Pascal Veron. HQC. https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/round-4/submissions/HQC-Round4.zip. Round 4 submission, 2022.

[84] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[85] Kaushik Nath and Palash Sarkar. Efficient arithmetic in (pseudo-)mersenne prime order fields. *Advances in Mathematics of Communications*, 16(2):303–348.

[86] H Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Probl. Control and Inform. Theory*, 15:19–34, 1986.

[87] NIST. Call for proposals. The Sage Development Team, https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf(accessed on 9th August, 2023), 2016.

[88] NIST. Round 4 submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions, 2023.

[89] Raphael Overbeck and Nicolas Sendrier. Code-Based Cryptography. https://www.researchgate.net/publication/226115302_Code-Based_Cryptography. In the book: Post-Quantum Cryptography (pp.95-145).

[90] Christiane Peters. Information-Set Decoding for Linear Codes over $\mathbf{F}_q$. In Nicolas Sendrier, editor, *Post-Quantum Cryptography*, pages 81–94, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[91] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Trans. Inf. Theory*, 8(5):5–9, 1962.

[92] Michael O. Rabin and Shmuel Winograd. Fast evaluation of polynomials by rational preparation. *Communications on Pure and Applied Mathematics*, 25(4):433–458, 1972.

[93] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[94] Palash Sarkar. Efficient Tweakable Enciphering Schemes from (Block-Wise) Universal Hash Functions. *IEEE Transactions on Information Theory*, 55(10):4749–4759, 2009.

[95] Palash Sarkar. A new multi-linear universal hash family. *Des. Codes Cryptography*, 69(3):351–367, dec 2013.

[96] R. Schroeppel and A. Shamir. A $T = \tilde{O}(2^{n/2})$, $S = \tilde{O}(2^{n/4})$ algorithm for certain NP-complete problems. In H. Gilbert, editor, *Annual International Conference on the Theory and Applications of Cryptographic Techniques EUROCRYPT 2010: Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science book series*. Springer.

[97] Nicolas Sendrier. Decoding One Out of Many. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, pages 51–67, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[98] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[99] Victor Shoup. On fast and provably secure message authentication based on universal hashing. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 313–328. Springer, 1996.

[100] Jerome A. Solinas. Generalized Mersenne Numbers. https://cacr.uwaterloo.ca/techreports/1999/corr99-39.pdf, 1999.

[101] Jacques Stern. A method for finding codewords of small weight. In Gérard D. Cohen and Jacques Wolfmann, editors, *Coding Theory and Applications, 3rd International Colloquium*, volume 388 of *Lecture Notes in Computer Science*, page 106–113. Springer, 1988.

[102] Jacques Stern. A New Identification Scheme Based on Syndrome Decoding. In *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 13–21. Springer, 1993.

[103] Richard Taylor. An Integrity Check Value Algorithm for Stream Ciphers. In Douglas R. Stinson, editor, *Advances in Cryptology — CRYPTO' 93*, pages 40–48, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[104] Thomas Johansson, and G. Kabatianskii and Ben Smeets. On the relation between A-codes and codes correcting independent errors. In *Advances in Cryptology / Lecture Notes in Computer Science*, volume 765, pages 1–11. Springer, 1993.

[105] Valentin Vasseur. Information set decoding implementation. `https://github.com/vvasseur/isd/blob/master/src/dumer.c` (accessed on 9th August, 2023), 2019.

[106] V.Sidelnikov and S.Shestakov. On insecurity of cryptosystems based on generalized Reed-Solomon codes. *Discrete Mathematics and Applications*, 2(4).

[107] David Wagner. A Generalized Birthday Problem. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 288–304, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[108] Floyd Zweydinger. Decoding. `https://github.com/FloydZ/decoding/blob/master/src/dumer.h` (accessed on 9th August, 2023), 2022.