

國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

快速常數時間模反元素演算法程式之形式驗證
Formal Verification of Fast Constant Time Modular
Inverse Algorithm Implementations

陳翰霆

Han-Ting Chen

指導教授: 陳偉松, 王柏堯 博士

Advisor: Tony Tan, Bow-Yaw Wang Ph.D.

中華民國 112 年 6 月

June, 2023

國立臺灣大學碩士學位論文
口試委員會審定書

MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

快速常數時間模反元素演算法程式之形式驗證

Formal Verification of Fast Constant Time Modular Inverse
Algorithm Implementations

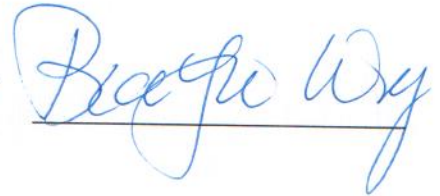
本論文係陳翰霆君（學號 R10922073）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 112 年 6 月 13 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 13 June 2023 have examined a Master's thesis entitled above presented by CHEN,HAN-TING (student ID: R10922073) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:



(指導教授 Advisor)



系主任/所長 Director:







Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Bow-Yaw Wang, for his unwavering support and invaluable guidance during the course of this research. His insightful advice and assistance have been instrumental in overcoming various challenges encountered along the way.

I am deeply thankful to Dr. Bo-Yin Yang, the inventor and programmer of the algorithm, for generously sharing his knowledge and providing a comprehensive understanding of the program. His expertise and explanations have greatly contributed to the success of this thesis.

I would also like to extend my appreciation to Zih-Ming Li, my research fellow, for his valuable contribution in providing the Coq proof, which has significantly strengthened the verification process.

Finally, I would like to thank my family for their support and care throughout my research work.





摘要

我們使用形式驗證工具 CRYPTO LINE 驗證兩個利用 Bernstein-Yang 演算法的模反元素程式，其中一個是目前以模數為 $2^{255} - 19$ 最快的 x86 實作。本論文提供了驗證此程式所用到的驗證細節與技巧。我們利用形式化方法驗證了此程式的正確性，也展現了一個形式驗證在證明密碼學系統的可信度上的應用。

關鍵字：形式驗證、模型檢測、模反元素、輾轉相除法、密碼學實作、Curve25519





Abstract

In this thesis, we conducted formal verification using the CRYPTOLINE tool on two x86 implementations of the Bernstein-Yang algorithm, both designed to operate in constant time. Notably, one of these implementations represents the current fastest constant time modular inversion implementation for prime modulus $2^{255} - 19$ on x86. Our study provides comprehensive details and verification techniques for verifying these assembly implementations. By formal methods, the correctness of these implementations is systematically demonstrated. The results of this study provide substantial evidence for the effectiveness of formal verification in ensuring the accuracy and reliability of cryptographic systems.

Keywords: formal verification, model checking, modular inversion, gcd, cryptographic programs, Curve25519





Contents

| | Page |
|---|-------------|
| Verification Letter from the Oral Examination Committee | i |
| Acknowledgements | iii |
| 摘要 | v |
| Abstract | vii |
| Contents | ix |
| List of Figures | xiii |
| List of Tables | xv |
| Denotation | xix |
| Chapter 1 Introduction | 1 |
| Chapter 2 Preliminary | 3 |
| 2.1 Modular Inverse Algorithms | 3 |
| 2.2 Original Bernstein-Yang Algorithm | 4 |
| 2.2.1 Definition of 2-adic division steps | 4 |
| 2.2.2 Iterations of 2-adic division steps | 5 |
| 2.2.3 Fast computation of iterations of 2-adic division steps | 8 |
| 2.2.4 Fast modular inversion computation | 9 |
| 2.3 Improved Bernstein-Yang Algorithm | 11 |



| | | |
|------------------|---|-----------|
| Chapter 3 | Introduction to Cryptoline | 13 |
| 3.1 | Why Formal Verification | 13 |
| 3.2 | What is CRYPTO LINE | 14 |
| 3.2.1 | Property | 15 |
| 3.2.2 | The structure of a CRYPTO LINE program | 18 |
| 3.2.3 | CRYPTO LINE instructions | 19 |
| 3.3 | Examples | 23 |
| 3.3.1 | Examples of modeling | 23 |
| 3.3.2 | CRYPTO LINE tricks | 25 |
| 3.3.3 | Verify an assembly program | 31 |
| Chapter 4 | Verifying a Simple Implementation | 33 |
| 4.1 | x86 25519 Implementation | 34 |
| 4.1.1 | C implementation of <code>fpinv25519.c</code> | 34 |
| 4.2 | Verify Simple Subroutines | 39 |
| 4.2.1 | Verify modular addition | 39 |
| 4.2.2 | Verify conditional modular negation | 40 |
| 4.2.3 | Verify signed multiplication with addition | 40 |
| 4.2.4 | Verify modular multiplication | 41 |
| 4.2.5 | Verify signed multi-limb multiplication with addition | 41 |
| 4.3 | Verify 62 divstep iterations | 43 |
| 4.3.1 | Pseudo code of the subroutine | 43 |
| 4.3.2 | Verify 1 divstep iteration | 44 |
| 4.3.3 | Model the subroutine | 49 |



| | | |
|------------------|--|-----------|
| 4.3.4 | Verify signed multi-limb multiplication and shift | 51 |
| 4.3.5 | Completeness of verification of the subroutine | 52 |
| 4.4 | Results | 53 |
| Chapter 5 | Verifying a Fast Vectorized Implementation | 55 |
| 5.1 | Vectorized x86 25519 Implementation | 56 |
| 5.1.1 | Outline of the assembly code | 57 |
| 5.2 | Verify 20 divstep iterations | 62 |
| 5.2.1 | An alternative definition of divstep | 63 |
| 5.2.2 | Verify each divstep iteration | 64 |
| 5.3 | Verify vectorized update | 67 |
| 5.3.1 | Pseudo code of the subroutine | 67 |
| 5.3.2 | Computing in parallel | 71 |
| 5.3.3 | Computing with Montgomery multiplication | 72 |
| 5.3.4 | Verify signed shift right computed with unsigned shift right | 74 |
| 5.3.5 | Use a proof from Coq | 75 |
| 5.3.6 | Reduce the output range | 76 |
| 5.4 | Verify radix 2^{30} number multiplication with reduction | 78 |
| 5.5 | Verify simple subroutines | 80 |
| 5.6 | Interleaving instructions | 82 |
| 5.7 | Results | 83 |
| Chapter 6 | Concluding Remarks | 85 |
| 6.1 | Time Consumption | 85 |
| 6.2 | The Verified Results | 86 |

References

Appendix A — Proof

A.1 Proof about arithmetic precision 89

Appendix B — Table

B.1 Verification Time of the Simple Implementation 91

B.2 Verification time of the Fast Vectorized Implementation 92





List of Figures





List of Tables

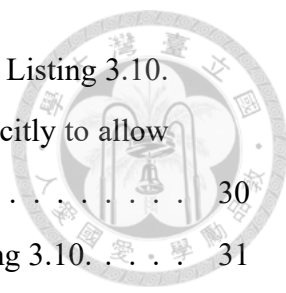
| | | |
|-----|--|----|
| B.1 | Verification time of the simple implementation. | 91 |
| B.2 | Verification time of the fast vectorized implementation. | 92 |





Listings

| | | |
|------|--|----|
| 2.1 | Algorithm <code>divsteps2</code> to compute $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$. Inputs: $n, t \in \mathbb{Z}$ with $0 \leq n \leq t$; $\delta \in \mathbb{Z}$; at least bottom t bits of $f, g \in \mathbb{Z}_2$. Outputs: δ_n ; bottom t bits of f_n if $n = 0$, or $t - (n - 1)$ bits if $n \geq 1$; bottom $t - n$ bits of g_n ; $\mathcal{T}_{n-1} \cdots \mathcal{T}_0$ | 7 |
| 2.2 | Algorithm <code>jumpdivsteps2</code> to compute $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$. Same inputs and outputs as in Listing 2.1. | 8 |
| 2.3 | Algorithm <code>recip2</code> to compute the reciprocal of g modulo f when $\text{gcd}\{f, g\} = 1$. The algorithm assumes that f is odd. | 10 |
| 3.1 | An example of a range property. | 16 |
| 3.2 | An example of an algebraic property. | 17 |
| 3.3 | Outline of a <code>CRYPTOLINE</code> program. | 18 |
| 3.4 | An example of <code>assert</code> and <code>assume</code> . <code>BOOLECTOR</code> will check whether $x = y$ during verification in the <code>assert</code> | 19 |
| 3.5 | A <code>vpc</code> example. | 21 |
| 3.6 | Outline of a <code>CRYPTOLINE</code> program that uses <code>call</code> | 22 |
| 3.7 | Some easy examples of modeling x86 instructions. | 24 |
| 3.8 | An example to model <code>cmp</code> and <code>cmovge</code> | 25 |
| 3.9 | An example of addition. $x, y \in \mathbb{Z}_{2^{255-19}}$ is stored in <code>limbs 64 [rdx, rcx, r8, rdi]</code> , and <code>limbs 64 [L0x7fffffffdd60, L0x7fffffffdd68, L0x7fffffffdd70, L0x7fffffffdd78]</code> respectively. After <code>Boolector</code> proves that the last carry is zero, this information can be assumed in <code>Singular</code> for later use. | 26 |
| 3.10 | Modeling <code>sar</code> instruction in <code>CRYPTOLINE</code> | 28 |



3.11 A multiplication instruction right after the sar instruction in Listing 3.10.
The property of the MSB of the multiplicand is stated explicitly to allow
SINGULAR to use later. 30

3.12 A bitwise AND instruction after the sar instruction in Listing 3.10. 31

4.1 `fpinv25519.c` implementation. Algorithm `fpinv25519` to compute the
reciprocal of g_0 modulo $2^{255} - 19$ when $\gcd\{2^{255} - 19, g_0\} = 1$ 36

4.2 Range property after the $\log_2 m'$ -th divstep iteration. 46

5.1 An example of simulating signed shift right using unsigned shift right. . . 75



Denotation

| | |
|----------|----------------------------------|
| ISA | Instruction set architecture |
| CF | Carry flag |
| SF | Sign flag |
| OF | Overflow flag |
| MSB | Most significant bit |
| CNF | Conjunctive normal form |
| DNF | Disjunctive normal form |
| SISD | Single instruction single data |
| SIMD | Single instruction multiple data |
| SSA form | Static single assignment form |
| AVX2 | Advanced Vector Extensions 2 |





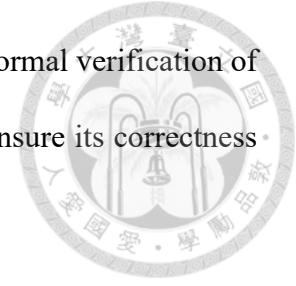
Chapter 1 Introduction

Computing modular inversion is a fundamental operation in cryptography that is frequently used in various cryptographic protocols. It involves finding the multiplicative inverse of an integer with respect to a given modulus. While the most common case is computing modular inversion over a prime modulus, the problem can be extended to arbitrary modular structures, such as modulo a polynomial. Generally, operations on secrets should be executed in constant time in cryptographic applications to avoid revealing information through side-channel attacks. Over the years, a variety of techniques have been proposed and refined to achieve efficient constant time algorithms for computing modular inversion, which are crucial for the practicality of the cryptographic systems.

Assessing the efficiency of an algorithm also involves evaluating the speed of its implementation since it is more practical to compare the execution time of an implementation. In cryptographic applications, arithmetic operations are often programmed in assembly language, and delicate programming techniques that leverage the strengths of a given platform are vital for achieving fast implementations.

When it comes to the implementation of a cryptographic systems, another important aspect is the correctness of the implementation. Unlike testing, where only a limited number of inputs can be validated, results from formal verification can be applied to all

possible inputs, making it a more trustworthy approach. Therefore, formal verification of an implementation is highly desirable in cryptographic systems to ensure its correctness and enhance its trustworthiness.



Organization of this thesis This thesis presents two case studies that focus on formal verification of fast modular inversion implementations. In Chapter 2, we introduce the algorithms used for computing modular inversion. In Chapter 3, we introduce the formal verification tool we use to verify arithmetic assembly programs. Chapter 4 presents a case study involving the verification of a simple implementation of modular inversion, while Chapter 5 presents another case study of the verification of the fastest implementation of modular inversion at the time. In Chapter 6, we describe the time consumption of the verification process and summarize the verification results.



Chapter 2 Preliminary

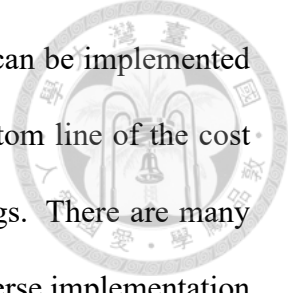
2.1 Modular Inverse Algorithms

To compute the multiplicative inverse in a modular structure, an essential algorithm is the extended Euclidean algorithm. It can be applied on the modular integers \mathbb{Z}_f where f is a positive integer and g is coprime with f . With f and g as the input to the algorithm, the algorithm computes the modular reciprocal $1/g \in \mathbb{Z}_f$. This algorithm can also be applied to the algebraic field extensions, which is typically common on polynomial quotient rings.

It seems like the extended Euclidean algorithm will be a good candidate to compute modular inversion. However, the Extended Euclidean algorithm and its variations are vulnerable to attacks in cryptography. The main reason is that these algorithms take variable time depending on the inputs. Oftentimes, the inputs are secret, and the difference in the execution time leaks information to the attacker through branch timing or cache timing. Therefore, we need constant time modular inverse algorithms. Here, constant time means that the execution time is independent of the inputs and therefore is fixed, which is different from the constant time complexity in the computational complexity theory.

A popular alternative is to compute by Fermat's little theorem. Given a finite field \mathbf{F}_p whose order is p , i.e., p is the smallest positive integer such that $a^p = a$ for all $a \in \mathbf{F}_p$.

We compute the inverse of a by $1/a = a^{p-2} \in \mathbf{F}_p$. This algorithm can be implemented in constant time, making it a safer choice in cryptography. The bottom line of the cost of a b -bit Fermat-style inversion is b full precision modular squarings. There are many decent results on various platforms of fast constant time modular inverse implementation via Fermat's little theorem.



Nevertheless, there are other algorithms that are even faster. In this chapter, we introduce some fast constant time alternatives to compute modular inversion.

2.2 Original Bernstein-Yang Algorithm

Bernstein-Yang Algorithm is introduced by Danial J. Bernstein and Bo-Yin Yang in 2019 [1]. This algorithm can be applied to compute not only in polynomial quotient rings with the x -adic division step but also in modular integers with the 2-adic division step. The paper also provided various algorithmic choices to use these division steps. In this section, we will focus on one specific variation which our fast x86 25519 implementation uses. Given a fixed odd integer f and an integer g which is coprime with f , this algorithm computes $1/g \in \mathbb{Z}_f$. Please consult [1] for the correctness of the algorithm. The implementation details will be provided in Section 4.1.

2.2.1 Definition of 2-adic division steps

Given an odd number f , we have the Bernstein-Yang 2-adic mapping as

$$\text{divstep} : (\delta, f, g) \mapsto \begin{cases} \left(\delta + 1, f, \frac{g+(g \bmod 2)f}{2} \right) & \text{if } \delta \leq 0 \text{ or } g \text{ is even} \\ \left(-\delta + 1, g, \frac{g-f}{2} \right) & \text{if } \delta > 0 \text{ and } g \text{ is odd} \end{cases} \quad (2.1)$$

Observe that $g + (g \bmod 2)f$ is always even and $g - f$ is even when g is odd.



2.2.2 Iterations of 2-adic division steps

Define

$$\mathcal{T}(\delta, f, g) = \begin{cases} \begin{bmatrix} 1 & 0 \\ \frac{g \bmod 2}{2} & \frac{1}{2} \end{bmatrix} & \text{if } \delta \leq 0 \text{ or } g \text{ is even} \\ \begin{bmatrix} 0 & 1 \\ \frac{-1}{2} & \frac{1}{2} \end{bmatrix} & \text{if } \delta > 0 \text{ and } g \text{ is odd} \end{cases}$$

$$\begin{bmatrix} f_{n+1} \\ g_{n+1} \end{bmatrix} = \mathcal{T}(\delta_n, f_n, g_n) \begin{bmatrix} f_n \\ g_n \end{bmatrix}.$$

Abbreviate $\mathcal{T}(\delta_n, f_n, g_n)$ by \mathcal{T}_n . The matrix $M_i(\delta_n, f_n, g_n)$ is defined as follows.

$$M_i(\delta_n, f_n, g_n) = \begin{cases} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \text{if } i = 0 \\ \mathcal{T}_{n+(i-1)} \mathcal{T}_{n+(i-2)} \cdots \mathcal{T}_n & \text{if } i > 0 \end{cases}$$

Let $\begin{bmatrix} u_i & v_i \\ r_i & s_i \end{bmatrix} = M_i(\delta_0, f_0, g_0)$. In Chapter 4 and Chapter 5 we will frequently use (u_i, v_i, r_i, s_i) to denote the computation that maps (δ_0, f_0, g_0) to (f_i, g_i) .

We can understand $M_i(\delta, f, g)$ as the computation of i divstep iterations on (δ, f, g) .

Thus, we can compute $n + i$ divstep iteration as:

$$\begin{bmatrix} f_{n+i} \\ g_{n+i} \end{bmatrix} = M_i(\delta_n, f_n, g_n) \begin{bmatrix} f_n \\ g_n \end{bmatrix}.$$



Also, we can compute $M_{i+j}(\delta_n, f_n, g_n)$ by:

$$M_{i+j}(\delta_n, f_n, g_n) = M_j(\delta_{n+i}, f_{n+i}, g_{n+i})M_i(\delta_n, f_n, g_n).$$

In the setup of computing modular inversion, $\delta_0 = 1, f_0 = f, g_0 = g$. This result shows that we can compute n divstep iterations by computing M_n . Notice that a part of the advantage of this algorithm is that when computing \mathcal{T} , only the bottom bits of f and g are used. Theorem 1 in Appendix A.1 shows the precision of f and g that is needed to compute M_n . This theorem implies that we can use lower precision to compute M_n . Then, whenever we want full precision f_n, g_n , just simply multiply M_n with full precision f, g to get the result.

Listing 2.1 shows a simple algorithm in Sage to compute n divstep iterations. This algorithm first uses `truncate` to reduce the precision of the loop variables `f, g` by cropping them into signed t -bit vectors and taking the values of the signed t -bit vectors. Then, initialize $(u, v, r, s) = (1, 0, 0, 1)$ and n divstep iterations will be computed in a loop. The i -th iteration computes \mathcal{T}_i and keeps the values of $M_i(\delta, f, g)$ in (u, v, r, s) . We use `ZZ` to convert the domain into the ring of arbitrary precision integers since the computation contains division. Notice that the values of f, g will always stay in the range of a signed t -bits vector throughout the loop¹. Finally, the output is returned and $M_n(\delta, f, g)$ is returned

¹This is equivalent to using full precision because of Theorem 1 in Appendix A.1.

as a 2×2 matrix over rational number by the conversion of `MatrixSpace(QQ,2)`.



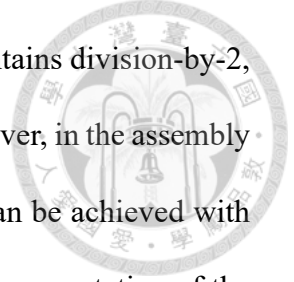
```

def truncate(f,t):
    if t == 0: return 0
    twot = 1<<(t-1)
    return ((f+twot)&(2*twot-1))-twot

def divsteps2(n,t,delta,f,g):
    assert t >= n and n >= 0
    f,g = truncate(f,t),truncate(g,t)
    u,v,r,s = 1,0,0,1
    while n > 0:
        f = truncate(f,t)
        if delta > 0 and g&1: delta,f,g,u,v,r,s = -delta,g
            ,-f,r,s,-u,-v
        g0 = g&1
        delta,g,r,s = 1+delta,(g+g0*f)/2,(r+g0*u)/2,(s+g0*v
            )/2
        n,t = n-1,t-1
        g = truncate(ZZ(g),t)
    M2Q = MatrixSpace(QQ,2)
    return delta,f,g,M2Q((u,v,r,s))

```

Listing 2.1: Algorithm `divsteps2` to compute $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$. Inputs: $n, t \in \mathbb{Z}$ with $0 \leq n \leq t$; $\delta \in \mathbb{Z}$; at least bottom t bits of $f, g \in \mathbb{Z}_2$. Outputs: δ_n ; bottom t bits of f_n if $n = 0$, or $t - (n - 1)$ bits if $n \geq 1$; bottom $t - n$ bits of g_n ; $\mathcal{T}_{n-1} \cdots \mathcal{T}_0$.



Notice that during the iterations, since the computation of \mathcal{T} contains division-by-2, to compute via M_n , the iterated variables u, v, r, s are fractions. However, in the assembly implementation, we want to use registers to store the values. This can be achieved with a twist to the algorithm. We can postpone the division by two in the computation of the transition matrix M_i . In other words, we compute $2\mathcal{T}$ instead of \mathcal{T} , so we will have $2^i M_i(\delta, f, g)$ after i -th iteration. In this way, $2^i M_i(\delta, f, g)$ will always hold integers so we can use registers to store the values. Notice that with this twist, we need an extra step of

division-by- 2^n to compute f_n, g_n as
$$\begin{bmatrix} f_n \\ g_n \end{bmatrix} = 2^n M_n(\delta, f, g) \begin{bmatrix} f \\ g \end{bmatrix} / 2^n.$$

2.2.3 Fast computation of iterations of 2-adic division steps

It is also possible to compute multiple divstep iterations by divide-and-conquer. In Listing 2.2, `jumpdivsteps2` is the algorithm to compute multiple divstep iterations with a divide-and-conquer method. Instead of directly computing $\text{divstep}^n(\delta, f, g)$, this algorithm first computes $\text{divstep}^j(\delta, f, g)$ to get δ_j and $M_j(\delta, f, g)$. Following this, it computes

$$\begin{bmatrix} f_j \\ g_j \end{bmatrix} = M_j(\delta, f, g) \begin{bmatrix} f \\ g \end{bmatrix}.$$
 After that, it computes the remaining $\text{divstep}^{n-j}(\delta_j, f_j, g_j)$ and merge the result.

```
from divsteps2 import divsteps2, truncate
```

```
def jumpdivsteps2(n, t, delta, f, g):
    assert t >= n and n >= 0
    if n <= 1: return divsteps2(n, t, delta, f, g)
```



```

j = n//2

delta,f1,g1,P1 = jumpdivsteps2(j,j,delta,f,g)

f,g = P1*vector((f,g))

f,g = truncate(ZZ(f),t-j),truncate(ZZ(g),t-j)

delta,f2,g2,P2 = jumpdivsteps2(n-j,n-j,delta,f,g)

f,g = P2*vector((f,g))

f,g = truncate(ZZ(f),t-n+1),truncate(ZZ(g),t-n)

return delta,f,g,P2*P1

```

Listing 2.2: Algorithm `jumpdivsteps2` to compute $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$. Same inputs and outputs as in Listing 2.1.

2.2.4 Fast modular inversion computation

Finally, We have the algorithm that computes multiplicative inverse, `recip2`, in Listing 2.3. Assuming f is an odd constant and $g \in \mathbb{Z}_f$ is coprime with f , by theorem², the reciprocal of g modulo f can be computed as

$$g^{-1} := (2^{m-1}v_m) * f_m * \left(\frac{1}{2}\right)^{m-1} \bmod f$$

where $(2^{m-1}v_m, f_m, (\frac{1}{2})^{m-1})$ are integers in \mathbb{Z}_f and $f_m \in \{1, -1\}$.

Integers(f)((f+1)/2)^(m-1) is used to compute the constant as $(\frac{1}{2})^{m-1} = (\frac{f+1}{2})^{m-1} \in$

²Theorem 11.2 in [1]

\mathbb{Z}_f . This constant can be pre-computed since it is independent to the input $g \in \mathbb{Z}_f$. `recip2` uses `divsteps2/jumpdivsteps2` as subroutines to compute divstep iterations.

In other words, this algorithm computes a fixed number of divstep iterations and multiplies the result with a pre-computed constant to get the modular inverse $g^{-1} \equiv v_m f_m \pmod{f}$.

```
from divsteps2 import divsteps2

def iterations(d):
    return (49*d+80)//17 if d<46 else (49*d+57)//17

def recip2(f,g):
    assert f & 1
    d = max(f.nbits(),g.nbits())
    m = iterations(d)
    precomp = Integers(f)((f+1)/2)^(m-1)
    delta, fm, gm, P = jumpdivsteps2(m,m+1,1,f,g)
    V = sign(fm)*ZZ(P[0][1]*2^(m-1))
    return ZZ(V*precomp)
```

Listing 2.3: Algorithm `recip2` to compute the reciprocal of g modulo f when $\gcd\{f, g\} = 1$. The algorithm assumes that f is odd.

2.3 Improved Bernstein-Yang Algorithm



This is a work by Daniel J. Bernstein, private communication and Pieter Wuille [2].

The new proof shows that if we start with $\delta_0 = \frac{1}{2}$ instead of $\delta_0 = 1$, less number of iterations are needed to get the modular inverse.

Take $f = 2^{255} - 19$ as an example, the original algorithm that starts with $\delta_0 = 1$ needs 738 iterations of divstep. With the new proof, only 600 divstep iterations are needed to compute the modular inverse if we start the algorithm with $\delta_0 = \frac{1}{2}$.

The improved algorithm is almost exactly the same as the original Bernstein-Yang algorithm, except for the starting parameters. Therefore, the definition in Section 2.2 can still be applied.





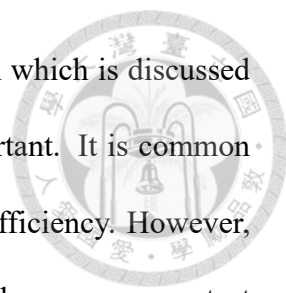
Chapter 3 Introduction to Cryptoline

3.1 Why Formal Verification

There are various approaches to verify whether a program behaves correctly as expected. Using test cases or testbench may be the simplest approach since the test engineers only need to plug in some different inputs and then check whether the outputs are correct. However, one weakness with testing is that oftentimes, the input space is so large that it will be infeasible to try every possible input during testing. Although there are indexes such as verification coverage to estimate the completeness of testing, there will always be some input cases left untested.

On the other hand, formal verification uses another approach. Instead of trying a number of test cases, it specifies the program as a mathematical model and uses mathematical checking to prove the properties of the model. Hence, formal verification can achieve full coverage, i.e., the result can be applied to all inputs.

Despite the advantages, formal verification is often unworkable due to the intensive work of modeling and mathematical checking. Domain experts are needed to give a model that captures all the important behaviors of the program. Moreover, proper mathematical checking algorithms are also needed in order to prove useful properties.



In cryptography, in addition to the robustness of a cryptosystem which is discussed in cryptanalysis, the correctness of the implementation is also important. It is common to implement arithmetic operations in cryptography in assembly for efficiency. However, assembly programs are complicated to understand. Even if the code passes some test cases, it is hard to make sure whether there are any hard-to-find bugs such as overflowing under some special inputs. That's why formal verification is worthwhile in this case. With formal verification, we can be confident that the program computes correctly under all possible inputs and that the outputs will not go out of their defined range.

3.2 What is CRYPTO LINE

CRYPTO LINE is a formal verification tool designed for assembly programmers to prove the properties of cryptographic assembly programs [4]. The modeling language, which is also called CRYPTO LINE, contains commonly used arithmetic instructions for cryptography. To transform an assembly program to its CRYPTO LINE model, the verification engineer will specify the semantics of each instruction to CRYPTO LINE by a script in the toolkit. Therefore, CRYPTO LINE is applicable for assembly programs in different instruction set architectures (ISAs).

The CRYPTO LINE language is typed. Every variable and constant is associated with its type in CRYPTO LINE. A type indicates the sign and the bit width of a variable or a constant. Type errors (such as subtracting an unsigned variable to a signed constant) in CRYPTO LINE programs are detected by the type system to reduce simple programming errors. Safety errors (such as adding two signed variable that causes overflow) are also detected.



3.2.1 Property

There are two kinds of properties in CRYPTO_{LINE}: range properties and algebraic properties.

1. Range property

A range property in CRYPTO_{LINE} is expressed as a propositional formula that supports logic operations $\{\wedge, \vee\}$. A propositional atom in a range property is the relation of equality, (signed/unsigned) greater than, (signed/unsigned) less than, or congruence modulo of variable expressions. A variable expression in a range property is composed of variables and constants with operations $\{+, -, \times, \text{slimbs}, \text{ulimbs}, \text{sext}, \text{uext}\}$ (sext/uext stands for signed/unsigned extension). The value of $(\text{slimbs } b [a_0, a_1, \dots, a_{(n-1)}])$ is defined as the value in $[-2^{nb-1}, 2^{nb-1} - 1]$ whose 2's complement representation equals the concatenation of $[a_0, a_1, \dots, a_{(n-1)}]$ (little-endian). Likewise, The value of $(\text{ulimbs } b [a_0, a_1, \dots, a_{(n-1)}])$ is defined as the value in $[0, 2^{nb} - 1]$ whose binary representation equals the concatenation of $[a_0, a_1, \dots, a_{(n-1)}]$ (little-endian). Notice that every variable expression in a range property is also associated with its bit width.

The CRYPTO_{LINE} tool verifies range properties by transforming each property to an instance of the satisfiability problem in the Quantified-Free Bit-Vector (QFBV) logic in Satisfiability Modulo Theories (SMT). The instance is then sent to the SMT QFBV solver BOOLECTOR [6]. A range property fails if and only if its instance is satisfiable. Because of the nature of BOOLECTOR, it is effective at recognizing the range of variables but struggles when the computation includes multi-limb multiplications.

Notice that with such transformation, the result proved by BOOLECTOR is still valid even if we disable the safety check. However, the overflow flag (OF) is not defined in CRYPTOLINE. Therefore, the verification engineers have to model the overflow flag manually if this information is needed. See Section 3.3.1 for examples.

```

or [and [rax = 0@64, rbx <s 20@64],
    eqmod rax 1@64 2@64,
    (sext rbx 64) = slimbs 64 [b0, b1]]

```

Listing 3.1: An example of a range property.

Listing 3.1 shows an example of expressing

$$((rax = 0) \wedge (rbx < 20)) \vee (rax \text{ is odd}) \vee (rbx = b_0 + b_1 * 2^{64})$$

as a range property in CRYPTOLINE where each variable is a signed 64-bit variable.

2. Algebraic property

Algebraic properties are introduced to cover the weakness of BOOLECTOR. An algebraic property in CRYPTOLINE is expressed as a propositional formula that only supports the logic operation $\{\wedge\}$. A propositional atom in an algebraic property is the relation of equality or congruence modulo of variable expressions. A variable expression in an algebraic property is composed of variables and constants with operations $\{+, -, \times, \text{limbs}\}$. Notice that a variable expression in an algebraic property does not have its bit width. Therefore, safety conditions must be passed to use the algebraic check in CRYPTOLINE safely due to the technical limitations of modeling. If a program overflows on some instructions and thus fails safety check, the

semantics of the program are still defined but cannot be captured by algebraic properties precisely. In this case, we usually constrain ourselves to use only the range properties to verify in order to maintain the soundness of the verification.

The CRYPTO LINE tool transforms each algebraic property to an instance of the ideal membership problem in commutative algebra. Given a polynomial and an ideal in a commutative ring, the ideal membership problem is to decide whether the polynomial belongs to the ideal. Instances of the ideal membership problem are solved by the computer algebra system SINGULAR [3]. With SINGULAR, CRYPTO LINE is capable of verifying computations with multi-limb multiplications efficiently.

```
and [eqmod limbs 64 [a0, a1]
      limbs 64 [b0, b1]
      f,
      (c - 1)*(c - 2)*(c - 3) = 0]
```

Listing 3.2: An example of an algebraic property.

Listing 3.2 shows an example of expressing

$$(a_0 + a_1 * 2^{64} \equiv b_0 + b_1 * 2^{64} \pmod{f}) \wedge ((c = 1) \vee (c = 2) \vee (c = 3))$$

as an algebraic property in CRYPTO LINE where each variable is a 64-bit variable.

With such transformation from range properties to instances for solver BOOLECTOR and algebraic properties to instances for solver SINGULAR in CRYPTO LINE, a property is verified if the underlying tool finds a proof for it. If a property fails verification, either the underlying tool finds a counterproof for it or the underlying tool does not halt in a

reasonable time limit.



3.2.2 The structure of a CRYPTO_{LINE} program

A CRYPTO_{LINE} program consists of a set of input parameters, the preconditions, the program body, and the postconditions. The following illustrates the purpose of each part:

1. Input parameters

A set of input variables. Each input variable is associated with its bit width and its type (signed or unsigned).

2. Preconditions

Specified the properties that hold before the execution of the program. The preconditions will be added as assumptions that hold unconditionally.

3. Program body

A block of CRYPTO_{LINE} instructions. See Section 3.2.3 for details.

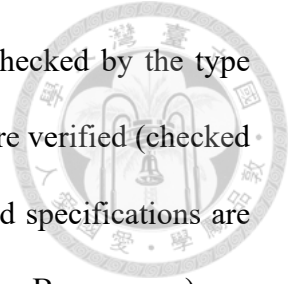
4. Postconditions

Specified the properties that hold after the execution of the program. The postconditions will be checked by the CRYPTO_{LINE} tool during verification.

```
proc main( INPUT_PARAMETERS )=  
{ PRECONDITION }  
  
PROGRAM_BODY  
  
{ POSTCONDITION }
```

Listing 3.3: Outline of a CRYPTO_{LINE} program.

A CRYPTO_{LINE} program is verified if: (a) it is well-formed (checked by the type system), (b) all the range properties in assertions and specifications are verified (checked by BOOLECTOR), and (c) all the algebraic properties in assertions and specifications are verified (checked by SINGULAR) and it passes safety check (checked by BOOLECTOR).



3.2.3 CRYPTO_{LINE} instructions

This Section describes the commonly used CRYPTO_{LINE} instructions:

1. Arithmetic instructions

CRYPTO_{LINE} supports a large number of commonly used arithmetic instructions for cryptography, including addition/subtraction(with/without carry/borrow), (signed/unsigned) multiplication, (signed/unsigned) shift, move, conditional move, logical operations such as and/or/not, data type conversions such as (signed/unsigned) split/join, etc. Notice that currently carry and borrow are the two flags that CRYPTO_{LINE} supports. See Section 3.3.1 for examples of when other flags are used in the computation.

2. Assertion

An assert instruction accepts two arguments: an algebraic property and a range property. Properties in assert instructions will be checked by SINGULAR or BOOLECTOR. Similar to the postcondition, except that assert instructions are added in the program body.

```
assert true && x = y;
```



```
assume x = y && true;
```

Listing 3.4: An example of `assert` and `assume`. `BOOLECTOR` will check whether $x = y$ during verification in the `assert`.



3. Assumption

For verification purposes, `CRYPTOLINE` also supports `assume` to add assumptions in the program body. An `assume` instruction also accepts two arguments: an algebraic property and a range property. Similar to the precondition, properties in `assume` instructions are assumed to hold unconditionally.

Listing 3.4 shows an example of using assumptions in `CRYPTOLINE`. This `assume` instruction can be safely added after the `assert` instruction. Typically, an `assume` instruction is added right after the property is proved using an `assert` by the other solver.

4. Cast/Value-preserving cast

Notice that every variable and constant is typed in `CRYPTOLINE`. Normally, programs use assembly instructions as the way it is designed. However, sometimes assembly programmers use instructions that are not well-formed but actually make sense in the context. Therefore, `CRYPTOLINE` supports `cast` to change the type of a variable, both its sign and bit width. Value-preserving cast (`vpc`) is also supported when the value fits the destination type.

Listing 3.5 shows a simple example of a value-preserving cast. The last addition would not be well-formed without the `vpc` instruction before it. Because the range of `rcx` falls in the range of a signed 64-bit vector $[-2^{63}, 2^{63} - 1]$, this `vpc` instruction will pass the safety check.



```
nondet rax@sint64; nondet rbx@uint64;  
assume true && and[(-50)@64 <s rax, rax <s 50@64];  
assume true && and[0@64 <=u rbx, rbx <=u 200@64];  
adds carry rcx rbx 3@uint64;  
vpc rcx@sint64 rcx;  
adds carry rdx rcx rax;
```

Listing 3.5: A vpc example.

5. Cut

When a CRYPTOLINE program is huge that it is difficult to verify as one single program, verification engineers often try to first break down the program into small pieces by their functionality to verify their computation, then connect the results from the verified pieces. The cut instruction is designed to meet the needs. A cut instruction literally cut a program into halves where each half is verified as an independent program. A cut instruction accepts two arguments: a range property and an algebraic property. The properties will be used as the postconditions for the first half and as the preconditions for the second half.

6. Call

CRYPTOLINE also supports function calls. `call` is the keyword for function call. The purpose is to simplify the process of verification by omitting to prove recurring codes. A `call` statement is composed of the function's identifier and a set of parameters. Similar to the structure of a CRYPTOLINE program in Section 3.2.2, the definition of a function contains the input/output parameters, the preconditions, the



function body, and the postconditions.

The current version of `call` in `CRYPTOLINE` is actually just a macro for inlining. In other words, it does not support local variables or other features for function calls. Therefore, verification engineers have to keep track of the naming of variables when using `call` in `CRYPTOLINE` to avoid conflicts.

However, `call` is still useful in many cases, it can not only be simply used as a macro for inlining a concrete subprogram but also serve as an abstraction layer to introduce the proved properties from a verified subprogram.

In the case of abstraction, which is the main use of `call`, the function body simulates a verified program. That is to say, the subprogram should be verified in a separate file. In the definition of such a function, the preconditions are asserted before the function body, since they should hold every time before calling the function. In the function body, the arithmetic instructions for computation are deleted, what remains is just some `nondet` instructions to declare every output variable as nondeterminate variables. After the function body, the postconditions are added as assumptions, which means the output variables are nondeterminate until the postconditions are assumed. Notice that the postconditions often state the exact relation between input and output variables, so no information should be lost in a perfectly successful verification.

```
proc functionid( INPUT/OUTPUT_PARAMETERS )=  
  { true && true }  
  assert PRECONDITION ;  
  FUNCTION_BODY  
  assume POSTCONDITION ;
```



```
{ true && true }

proc main( INPUT_PARAMETERS )=
{ PRECONDITION }
PROGRAM_BODY
...
call functionid( INPUT/OUTPUT_PARAMETERS );
...
PROGRAM_BODY
{ POSTCONDITION }
```

Listing 3.6: Outline of a CRYPTO_{LINE} program that uses call.

3.3 Examples

CRYPTO_{LINE} is capable of verifying assembly programs on various platforms. In this Section, we use x86 ISA in 64-bit mode to illustrate how to use CRYPTO_{LINE}.

3.3.1 Examples of modeling

Constants and variables Constants can be simply modeled by constants in CRYPTO_{LINE}. A register or an effective address is modeled by a variable in CRYPTO_{LINE}. In this way, we won't need to care about pointers or pointer aliasing, which simplifies verification.

Every register or effective address should be 64-bit wide at the start of the program.

It is the verification engineer's responsibility to decide the sign of a variable or a constant base on the value it represents.



Simple instructions For x86 instructions that CRYPTO_{LINE} has a semantically equivalent instruction, we simply model it with the corresponding CRYPTO_{LINE} instruction.

In Listing 3.7, `imul` with one operand `$1v` computes signed multiplication with `rax`, and stores the higher signed 64-bit result in `rdx`, lower unsigned 64-bit result in `rax`. It is simply modeled with a CRYPTO_{LINE} instruction `smull`.

```
#! add $1v, $2v -> adds carry $2v $1v $2v
#! adc $1v, $2v -> adcs carry $2v $1v $2v carry
#! imul $1v -> smull rdx rax $1v rax
```

Listing 3.7: Some easy examples of modeling x86 instructions.

Tricky Instructions To model an instruction that uses flags or instructions that CRYPTO_{LINE} does not directly support, we use multiple CRYPTO_{LINE} instructions to simulate such an instruction. The verification engineers are responsible for the semantic equivalence of the modeling.

Conditional Move Greater or Equal (`cmovge`) is one of the conditional move instructions in x86. Its semantics is defined by move if the sign flag (SF) equals the overflow flag (OF). A `cmovge` instruction usually appears after a `cmp` instruction where `cmp` subtracts its two operands and set the status flags in the EFLAGS register according to the result of the subtraction. Notice that if the relation between the two operands is greater than or equal to, the SF of the subtraction result will be equal to the OF of the subtraction result.

Listing 3.8 shows an example. A computation that uses a `cmp` with zero followed by `cmovge` is modeled in CRYPTO`LINE` with a subtraction-with-borrow (`subb`) to compute whether SF should be equal to OF. The result is kept in a bit `SFe0F` which is used as the flag for the conditional move instruction (`cmov`) in CRYPTO`LINE`.

```
#! cmp \ $0, $1v -> subb SFe0F dontcare $1v (-(2)**(63))
@sint64
#! cmovge $1v, $2v -> cmov $2v SFe0F $1v $2v
```

Listing 3.8: An example to model `cmp` and `cmovge`.

3.3.2 CRYPTO`LINE` tricks

With the above modeling techniques, we can now model an assembly program in CRYPTO`LINE`. However, if one just simply tries to verify the postcondition on this model, often the time it will not work. This is because both BOOLECTOR and SINGULAR are not strong enough on their own, we usually have to use one to prove some kinds of properties, the other to prove the rest.

Carry bit Usually, we need to manually transfer information between solvers. A common example is the carry bit: since BOOLECTOR holds information about the range of variables while SINGULAR doesn't, most of the time only BOOLECTOR can prove whether the value of a carry bit is zero. To transfer this information to SINGULAR, we first use an `assert` in BOOLECTOR and then an `assume` in SINGULAR of the same expression. With this `assume`, SINGULAR can use this information in the rest of the program.

Listing 3.9 is an easy example from the subroutine of modular addition in $\mathbb{Z}_{2^{255}-19}$. Since we are using 4 64-bit registers to store the input which is less than $2^{255} - 19$, the last carry bit after summing up the two inputs will be zero, which is proved by BOOLECTOR. After assuming this in SINGULAR, we can continue to prove the rest of the program.

```

adds carry rdx L0x7fffffffdd60 rdx;
adcs carry rcx L0x7fffffffdd68 rcx carry;
adcs carry r8 L0x7fffffffdd70 r8 carry;
adcs carry rdi L0x7fffffffdd78 rdi carry;

assert true && carry = 0@1;

assume carry = 0 && true;

```

Listing 3.9: An example of addition. $x, y \in \mathbb{Z}_{2^{255}-19}$ is stored in limbs 64 [rdx, rcx, r8, rdi], and limbs 64 [L0x7fffffffdd60, L0x7fffffffdd68, L0x7fffffffdd70, L0x7fffffffdd78] respectively. After Boolector proves that the last carry is zero, this information can be assumed in Singular for later use.

Assuming a CNF formula proved by BOOLECTOR in SINGULAR An arbitrary CNF formula can be transformed into a DNF formula via the schoolbook transformation where the length of the resulting DNF formula grows linearly with the size of the CNF but exponentially with the size of the clause of the CNF formula.

Sometimes during verification, a range property in CNF is proved by BOOLECTOR and we want to assume this property as an algebraic property in SINGULAR. However, it's not simple to express a CNF formula in SINGULAR, but it's straightforward to assume a DNF in SINGULAR. So we can first transform the CNF into a DNF and then assume the equivalent DNF in SINGULAR. Notice that we usually do this trick when the size of the CNF is small which is often the case. We don't recommend this trick when the size of the

CNF is so large that it will not be practical to assume the resulting DNF in SINGULAR.



For example, let's say the following CNF is proved by BOOLECTOR:

$\text{or}[\text{and}[\text{case1}, \text{property1}], \text{and}[\text{case2}, \text{property2}], \text{and}[\text{case3}, \text{property3}]]$

We manually transform this CNF into a DNF via schoolbook transformation:

$$(\text{case1} \vee \text{case2} \vee \text{case3}) \wedge (\text{case1} \vee \text{case2} \vee \text{property3}) \wedge$$

$$(\text{case1} \vee \text{property2} \vee \text{case3}) \wedge (\text{case1} \vee \text{property2} \vee \text{property3}) \wedge$$

$$(\text{property1} \vee \text{case2} \vee \text{case3}) \wedge (\text{property1} \vee \text{case2} \vee \text{property3}) \wedge$$

$$(\text{property1} \vee \text{property2} \vee \text{case3}) \wedge (\text{property1} \vee \text{property2} \vee \text{property3})$$

With the transformation, we can easily assume this DNF in SINGULAR as:

```
assume (case1 * case2 * case3 = 0) && true;
```

```
assume (case1 * case2 * property3 = 0) && true;
```

```
assume (case1 * property2 * case3 = 0) && true;
```

```
assume (case1 * property2 * property3 = 0) && true;
```

```
assume (property1 * case2 * case3 = 0) && true;
```

```
assume (property1 * case2 * property3 = 0) && true;
```

```
assume (property1 * property2 * case3 = 0) && true;
```



```
assume (property1 * property2 * property3 = 0) && true;
```



Signed multi-limb multiplication To multiply two 128-bit signed integers, a program loads each input into two 64-bit registers and combines the use of signed/unsigned multiplication instructions. To verify this computation, we need to specify the behavior of nonlinear instructions to SINGULAR manually.

Listing 3.10, 3.11, and 3.12 shows the important parts of an example of a program that computes signed multi-limb multiplication.

In Listing 3.10, the instruction `sar $0x3f, %r11`, whose semantics is signed division `r11` by two 63 times, is used to extend the sign bit of `r11` to a 64-bit mask. To translate this instruction, We extract the MSB and simulate the `sar` instruction with a subtraction and a conditional move. New variables: `msb`, `flag` are created as auxiliary terms. After BOOLECTOR proves its semantics with `assert`, the information is transferred to SINGULAR by the use of `assume`. Similar to the previous example, the range property proved by BOOLECTOR is a small CNF and we assume the algebraic property in SINGULAR as a DNF.

```
(* mov %rax,%r11 *)
mov r11 rax; (* r11 = rax with type sint64 *)
(* sar $0x3f,%r11 *)
and msb@uint64 r11 0x8000000000000000@uint64;
subc flag dontcare msb 1@uint64;
cmov r11 flag 0xffffffffffffffff@uint64 0@uint64;
assert true &&
    or[and[flag = 0@1, r11 = 0@64],
        and[flag = 1@1, r11 = 0xffffffffffffffff@64]];
```

```

assume (r11) * (flag - 1) = 0 && true;
assume (r11 - 0xffffffffffffffff) * flag = 0 && true;

```

Listing 3.10: Modeling sar instruction in CRYPTOLINE.




Right after the sar instruction, we have the the unsigned multiplication instruction in Listing 3.11. A normal unsigned multiplication computes multiplication of its two operands in $[0, 2^{64} - 1]$, and the put 128-bit result in $[0, 2^{128} - 2^{65} + 1]$ in ulimbs 64 $[rax, rdx]$. In this case, we want to use an unsigned multiplication to compute multiplication on a signed operand rax and an unsigned operand r12. This instruction will not be well-formed if we don't specify some additional information in CRYPTOLINE. Therefore, before the unsigned multiplication, a cast from signed to unsigned is used to change the type of rax. We keep the value of rax as sint64 in raxo for verification purposes. The cast will not affect BOOLECTOR since the bit width stays the same, but in SINGULAR, the value of rax after the cast will become $raxo + 2^{63} * dontcare$ for soundness because it is not a value-preserving cast.

However, we know that for a 64-bit vector whose MSB is $high \in [0, 1]$ and the value of its lower 63 bits as an unsigned 63-bit vector is $low \in [0, 2^{63} - 1]$, if we treat it as a signed bit vector, its value will be $-2^{63} * high + low \in [-2^{63}, 2^{63} - 1]$. On the other hand, if we treat it as an unsigned vector, its value will be $2^{63} * high + low \in [0, 2^{64} - 1]$.

This is what we are going to specify in CRYPTOLINE, because the value of the MSB of rax after the cast should not be *dontcare* in this case. We need to explicitly specify that the value of the MSB is negated after a cast from signed to unsigned. Moreover, we also specify the fact that the MSB of rax equals the flag in the simulation of sar in Listing 3.10. Finally, after the unsigned multiplication instruction, we explicitly specify that the

result of the multiplication can also be represented by `raxh` and `raxl` instead only by `rax`.



```
(* mul %r12 *)
mov raxo rax; (* The type of rax is sint64 *)
cast rax@uint64 rax;
usplit raxh raxl rax 63;
assert true && raxh * const 64 (-(2**63)) + raxl = raxo;
assume raxl - raxh * (2**63) = raxo && true;

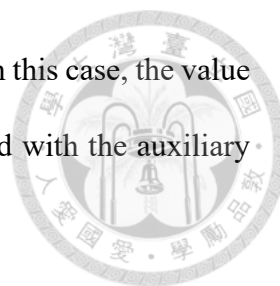
assert true && (uext flag 63) = raxh;
assume flag = raxh && true;

umull rdx rax r12 rax;

assert true && limbs 64 [rax, rdx] = (uext r12 64) * ((uext
    raxh 64) * (const 128 (2**63)) + (uext raxl 64));
assume limbs 64 [rax, rdx] = r12 * (raxh * 2**63 + raxl) &&
    true;
```

Listing 3.11: A multiplication instruction right after the `sar` instruction in Listing 3.10. The property of the MSB of the multiplicand is stated explicitly to allow SINGULAR to use later.

The bitwise AND instruction in Listing 3.12 shows the usage of the mask computed by `sar` in Listing 3.10. The auxiliary variable `flag` indicated the computed mask in `r11` should be an all-zero or all-one vector. Because we cannot express the computation of bitwise AND as an equation, so we don't have an algebraic model for such an operation



in CRYPTO_{LINE}. The verification engineer has to model it manually. In this case, the value after the bitwise AND with the mask in r11 can be simply expressed with the auxiliary variable flag.

```
(* and %r11,%r12 *)  
  
mov r12o r12;  
  
and r12@uint64 r11 r12;  
  
assert true && or[and[flag = 0@1, r12 = 0@64],  
                and[flag = 1@1, r12 = r12o]];  
  
assume (flag) * (r12 - r12o) = 0 && true;  
  
assume (flag - 1) * (r12) = 0 && true;
```

Listing 3.12: A bitwise AND instruction after the sar instruction in Listing 3.10.

Without these tricks, the algebraic properties of the multi-limb multiplication in the postcondition would not be provable since there will be some lost information to SINGULAR. Here, We omit the rest of the computations. For the details of the verification of a concrete multi-limb multiplication subroutine, please see the verification code at https://github.com/fmlab-iis/cryptoline/blob/master/examples/ct_inverse/bernstein_yang/25519/x86/mul2x2s128_25519.cl.

3.3.3 Verify an assembly program

See Chapter 4 and 5 for concrete examples of the verification process of actually used assembly subroutines.





Chapter 4 Verifying a Simple Implementation

We can only discuss the asymptotic computational time complexity of an algorithm if it is a pseudo algorithm. In practice, the platform support and the efficiency of the implementation are also crucial when discussing speed. In cryptography, fast implementation is already a popular topic. A fast implementation will be more valuable if its correctness is guaranteed.

In this chapter, we show a case study of the verification of a fast implementation of the original Bernstein-Yang algorithm. Section 4.1 describes a simple (but still pretty fast) implementation of the original Bernstein-Yang algorithm. Section 4.2 and 4.3 show how we use CRYPTOLINE to verify this implementation. Section 4.4 describes the verification result, and how is it related to the correctness of the implementation. The implementation and the verification code can be found at https://github.com/fmlab-iis/cryptoline/tree/master/examples/ct_inverse/bernstein_yang/25519/x86.



4.1 x86 25519 Implementation

This is an x86 implementation of the 2-adic version of the original Bernstein-Yang algorithm with $f_0 = 2^{255} - 19$ written by Bo-Yin Yang. It takes 8778 cycles on Intel Skylake CPU core, which is decent compared to the state-of-the-art. By theorem ¹, 744 iterations is enough to compute the multiplicative inverse with $f_0 = 2^{255} - 19$. We specify the C implementation of this algorithm in Listing 4.1. This implementation uses divide-and-conquer to compute divstep^{744} to simulate `recip2` in Listing 2.3. All of the computation in this C function is implemented with some subroutines written in assembly. These subroutines in assembly use some counterintuitive tricks to compute efficiently. This makes the program difficult to read or debug. To boost the confidence of the program, we use CRYPTOLINE to verify the subroutines. We will give more details about the implementation of these assembly subroutines while explaining how we verify them in Section 4.2 and 4.3.

4.1.1 C implementation of `fpinv25519.c`

Following the notation in Chapter 2, we use the notation

$$(\delta_i, f_i, g_i) = \text{divstep}^i(\delta_0, f_0, g_0)$$

and

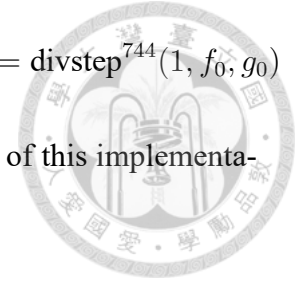
$$\begin{bmatrix} u_i & v_i \\ r_i & s_i \end{bmatrix} = M_i(\delta_0, f_0, g_0)$$

in this chapter.

Remember that by the algorithm `recip2` in Listing 2.3, we can get the modular in-

¹Theorem 11.2 in [1].

verse by computing $g_0^{-1} = f_{744}v_{744} \pmod{f_0}$ where $(\delta_{744}, f_{744}, g_{744}) = \text{divstep}^{744}(1, f_0, g_0)$ and $\begin{bmatrix} u_{744} & v_{744} \\ r_{744} & s_{744} \end{bmatrix} = M_{744}(1, f_0, g_0)$. Therefore, we can say the goal of this implementation is to compute $f_{744}v_{744} \pmod{f_0}$.



Before looking into the subroutines in assembly, let's first look at the C function that computes modular inversion. We give the C code in Listing 4.1. Please refer to Section 4.2 and 4.3 for the definitions of the subroutines.

The function `fpinv25519` uses divide-and-conquer to compute divstep^{744} to simulate `recip2` in Listing 2.3. It starts with $\delta_0 = 1$, $f_0 = 2^{255} - 19$ and g_0 as the input in $[-2^{255}, 2^{255} - 1]$.

The first call to `jump64divsteps2_s255` computes $\delta_{62}, f_{62}, g_{62}$ and $2^{62} * M_{62}(\delta_0, f_0, g_0)$ in `b`. The second call to `jump64divsteps2_s255` computes $\delta_{62*2}, f_{62*2}, g_{62*2}$ and $2^{62} * M_{62}(\delta_{62}, f_{62}, g_{62})$ in `b2`. To compute $2^{62*2} * M_{62*2}(\delta_0, f_0, g_0)$, it simply uses `mul64xs64` to compute the 2×2 to 2×2 matrix product on `b` and `b2` and stores the result on `b3`. The same technique is used to compute $\delta_{62*4}, f_{62*4}, g_{62*4}$ and $2^{62*2} * M_{62*2}(\delta_{62*2}, f_{62*2}, g_{62*2})$ in `b4`. To compute $2^{248} * M_{248}(\delta_0, f_0, g_0)$, it uses `mul2x2s128_25519` to compute the 2×2 to 2×2 matrix product on `b3` and `b4` and stores the result in `b5`. That sums up the computation of the first 248 iterations.

Starting with $\delta_{248}, f_{248}, g_{248}$, we use the same technique to compute $\delta_{248*2}, f_{248*2}, g_{248*2}$ and $2^{248} * M_{248}(\delta_{248}, f_{248}, g_{248})$ in the second 248 iterations. Here, we use `fpmul2x2_25519_half` to compute 2×2 to 2×2 matrix product in $\mathbb{Z}_{2^{255}-19}$, but we only compute the right half of the resulting 2×2 matrix. `fpmul2x2_25519_half` is implemented with some calls to `fpmul25519` and `fpadd25519` for modular multiplication and modular addition in $\mathbb{Z}_{2^{255}-19}$. Consequently, we have the right half of $2^{248*2} * M_{248*2}(\delta_0, f_0, g_0)$ in `b7` which

is $2^{248*2} * v_{248*2}$ and $2^{248*2} * s_{248*2}$ before the third 248 iterations.

Again, with $\delta_{248*2}, f_{248*2}, g_{248*2}$, we compute $2^{248*3} * M_{248}(\delta_{248*2}, f_{248*2}, g_{248*2})$ in b6 with the same technique. To compute the upper right entry of $2^{248*3} * M_{248*3}(\delta_0, f_0, g_0)$ which is $2^{248*3} * v_{248*3}$, we use `fpmul2x2_25519_quarter` to compute the upper right entry of the 2×2 to 2×2 matrix product on b6 and b7. `fpmul2x2_25519_quarter` is also implemented with calls to `fpmul25519` and `fpadd25519` for modular multiplication and modular addition in $\mathbb{Z}_{2^{255}-19}$.

Finally, with $2^{744} * v_{744}$, we compute g_0^{-1} by $g_0^{-1} = 2^{744} * v_{744} * (\frac{1}{2})^{744} * f_{744} \in \mathbb{Z}_{2^{255}-19}$. It uses `fpmul25519` for the modular multiplication with the pre-computed constant `inv2744 = (\frac{1}{2})^{744}` and `fpcneg25519` for the modular negation using `f_{744}` as the flag (`f_{744}` is guaranteed to be 1 or -1 by theorem, so `f[3] = f_{744}`).

```
void fpmul2x2_25519_quarter(unsigned long long m1[16],
    unsigned long long m2[16], unsigned long long m3[16]) {
    unsigned long long t0[4], t1[4];
    fpmul25519(m1, m2+4, t0);
    fpmul25519(m1+4, m2+12, t1);
    fpadd25519(t0, t1, m3+4);
}

void fpmul2x2_25519_half(unsigned long long m1[16],
    unsigned long long m2[16], unsigned long long m3[16]) {
    unsigned long long t0[4], t1[4];
    fpmul25519(m1, m2+4, t0);
    fpmul25519(m1+4, m2+12, t1);
    fpadd25519(t0, t1, m3+4);
}
```



```
fpmul25519(m1+8,m2+4,t0);
fpmul25519(m1+12,m2+12,t1);
fpadd25519(t0,t1,m3+12);
}

void fpinv25519(unsigned long long g[4],
               unsigned long long inv_g[4]) {
    int delta;
    unsigned long long inv2744[4] = {0x5dc1855b1b224df9,
                                     0x9ca54469d9422c90, 0x59639d9db0ccd471,
                                     0x38b66f98b076d64f};
    unsigned long long f[4] = {-19LL, -1LL, -1LL,
                               0x7fffffffffffffffffULL};
    unsigned long long b[4], b2[4];
    unsigned long long b3[8], b4[8];
    unsigned long long b5[16], b6[16], b7[16];

    // start of first 248 iterations
    delta = jump64divsteps2_s255(62,1,f,g,b);
    delta = jump64divsteps2_s255(62,delta,f,g,b2);
    muls64xs64(b2,b,b3);
    delta = jump64divsteps2_s255(62,delta,f,g,b);
    delta = jump64divsteps2_s255(62,delta,f,g,b2);
    muls64xs64(b2,b,b4);
```



```
mul2x2s128_25519(b4,b3,b5);

// start of second 248 iterations

delta = jump64divsteps2_s255(62,delta,f,g,b);
delta = jump64divsteps2_s255(62,delta,f,g,b2);
muls64xs64(b2,b,b3);
delta = jump64divsteps2_s255(62,delta,f,g,b);
delta = jump64divsteps2_s255(62,delta,f,g,b2);
muls64xs64(b2,b,b4);
mul2x2s128_25519(b4,b3,b6);
fpmul2x2_25519_half(b6,b5,b7);

// start of third 248 iterations

delta = jump64divsteps2_s255(62,delta,f,g,b);
delta = jump64divsteps2_s255(62,delta,f,g,b2);
muls64xs64(b2,b,b3);
delta = jump64divsteps2_s255(62,delta,f,g,b);
delta = jump64divsteps2_s255(62,delta,f,g,b2);
muls64xs64(b2,b,b4);
mul2x2s128_25519(b4,b3,b6);
fpmul2x2_25519_quarter(b6,b7,b5);

fpmul25519(b5+4,inv2744,inv_g);
fpcneg25519(inv_g,f[3]);
```

}

Listing 4.1: `fpinv25519.c` implementation. Algorithm `fpinv25519` to compute the reciprocal of g_0 modulo $2^{255} - 19$ when $\gcd\{2^{255} - 19, g_0\} = 1$.



4.2 Verify Simple Subroutines

This section describes how we verify the implementation while illustrating how assembly programs are verified in CRYPTO_{LINE}.

The translating rules from x86 instructions to CRYPTO_{LINE} instructions as mentioned in Section 3.3.1 is put with each of the code of the subroutines in a separated `.gas` format file. With the `.gas` file as the input, the `to_zds1.py` script in the CRYPTO_{LINE} toolkit will generate the corresponding `.c1` format file which is the model of assembly code in CRYPTO_{LINE}. Then, we manually specify the sign and the bit width of each input variable, precondition, and postcondition in the `.c1` file. Finally, we can start to try to verify them. The following subsections describe how we verify each of the assembly subroutines in Section 4.1.

4.2.1 Verify modular addition

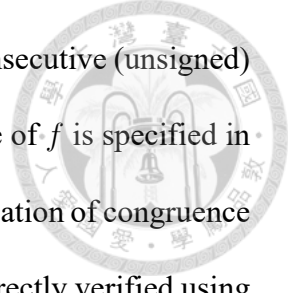
- `void fpadd25519(unsigned long long a[4], unsigned long long b[4], unsigned long long r[4]);`

Input: $a, b \in \mathbb{Z}_{2^{255}-19}$;

Output: $r = a + b \in \mathbb{Z}_{2^{255}-19}$.

`fpadd25519` is the subroutine that computes modular addition. The input a, b are

unsigned 256-bit vectors in $[0, 2^{255} - 20]$. Each input is stored in 4 consecutive (unsigned) 64-bit locations in the memory. The range of the inputs and the value of f is specified in the precondition in BOOLECTOR. The postcondition is specified as a relation of congruence modulo in BOOLECTOR. This is a rather simple example and can be directly verified using only range properties this way.



4.2.2 Verify conditional modular negation

- `void fpcneg25519(unsigned long long a[4], long long flag);`

Input: $a \in \mathbb{Z}_{2^{255}-19}$; $flag \in [-2^{63}, 2^{63}]$;

Output: $a \in \mathbb{Z}_{2^{255}-19}$ if $flag \geq 0$; $-a \in \mathbb{Z}_{2^{255}-19}$ if $flag < 0$.

`fpcneg25519` is the subroutine that computes conditional modular negation. Similar to Subsection 4.2.1, this subroutine is easily verified by BOOLECTOR.

4.2.3 Verify signed multiplication with addition

- `void muls64xs64(unsigned long long X[4], unsigned long long Y[4],`

`unsigned long long M[8]);`

Input: $\begin{bmatrix} x_0 & x_1 \\ x_2 & x_3 \end{bmatrix}, \begin{bmatrix} y_0 & y_1 \\ y_2 & y_3 \end{bmatrix}$ where $x_i, y_i \in [-2^{62}, 2^{62}]$;

Output: $\begin{bmatrix} m_0 & m_1 \\ m_2 & m_3 \end{bmatrix} = \begin{bmatrix} x_0 & x_1 \\ x_2 & x_3 \end{bmatrix} \begin{bmatrix} y_0 & y_1 \\ y_2 & y_3 \end{bmatrix}$ where $m_i \in (-2^{125}, 2^{125})$.

`muls64xs64` is the subroutine that computes signed multiplication with addition.

Similar to Subsection 4.2.1, this subroutine is easily verified by BOOLECTOR.



4.2.4 Verify modular multiplication

- `void fpmul25519(unsigned long long a[4], unsigned long long b[4], unsigned long long r[4]);`

Input: $a, b \in \mathbb{Z}_{2^{255}-19}$;

Output: $r = ab \in \mathbb{Z}_{2^{255}-19}$.

`fpmul25519` is the subroutine that modular multiplication. Unlike the case in Subsection 4.2.1, 4.2.2, or 4.2.3, this computation contains unsigned multi-limb multiplication which is one of the weaknesses of `BOOLECTOR`. Therefore, we use `SINGULAR` to verify this computation. As mentioned in Section 3.3.2, we use `assert/assume` to tell `SINGULAR` that some registers/carry bits should be zero during the computation. Also, we use `BOOLECTOR` to check the range of the modular reduction. With these techniques, we successfully verified the postcondition of this subroutine.

4.2.5 Verify signed multi-limb multiplication with addition

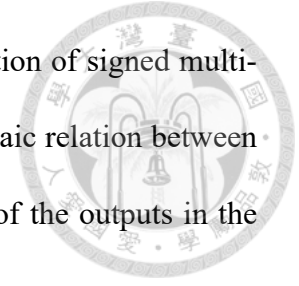
- `void mul2x2s128_25519(unsigned long long X[8], unsigned long long Y[8], unsigned long long M[16]);`

Input: $\begin{bmatrix} x_0 & x_1 \\ x_2 & x_3 \end{bmatrix}, \begin{bmatrix} y_0 & y_1 \\ y_2 & y_3 \end{bmatrix}$ where $x_i, y_i \in [-2^{125} + 1, 2^{125} - 1]$;

Output: $\begin{bmatrix} m_0 & m_1 \\ m_2 & m_3 \end{bmatrix} = \begin{bmatrix} x_0 & x_1 \\ x_2 & x_3 \end{bmatrix} \begin{bmatrix} y_0 & y_1 \\ y_2 & y_3 \end{bmatrix}$.

`mul2x2s128_25519` is the subroutine that computes multi-limb signed multiplication with addition. This subroutine also takes some effort to verify. We use the trick

mentioned in Section 3.3.2 to specify the properties of the computation of signed multi-limb multiplication explicitly. We use SINGULAR to verify the algebraic relation between the inputs and the outputs and use BOOLECTOR to verify the range of the outputs in the postcondition.





4.3 Verify 62 divstep iterations

4.3.1 Pseudo code of the subroutine

To explain the computation of this subroutine, we provide the pseudo code in Algorithm 1. We use \bar{x} to denote the truncated value of x in a lower precision.

Algorithm 1 Pseudo jump64divsteps2_s255

Input: $\delta \in [-2^{60}, 2^{60}]$, $f, g \in \mathbb{Z}_{2^{255}-19}$.

Output: $(\delta'_{62}, f'_{62}, g'_{62}) = \text{divstep}^{62}(\delta, f, g)$; $\begin{bmatrix} u'_{62} & v'_{62} \\ r'_{62} & s'_{62} \end{bmatrix} = 2^{62}M_{62}(\delta, f, g)$.

- 1: $(\bar{f}, \bar{g}) \leftarrow (\text{truncate}(f, 64), \text{truncate}(g, 64))$
 - 2: $(\hat{f}, \hat{g}) \leftarrow (\bar{f}, \bar{g})$ ▷ Saved for verification purpose.
 - 3: $(u, v, r, s) \leftarrow (1, 0, 0, 1)$
 - 4: **for** $i \leftarrow 1$ to 31 **do**
 - 5: $(\delta', \bar{f}', \bar{g}', u', v', r', s') \leftarrow \text{DivstepImplementation1}(\delta, \bar{f}, \bar{g}, u, v, r, s)$
 - 6: $(\delta, \bar{f}, \bar{g}, u, v, r, s) \leftarrow \text{DivstepImplementation2}(\delta', \bar{f}', \bar{g}', u', v', r', s')$
 - 7: $(u'_{62}, v'_{62}, r'_{62}, s'_{62}) \leftarrow (u, v, r, s)$
 - 8: $\delta'_{62} \leftarrow \delta$
 - 9: $f'_{62} \leftarrow (u'_{62}f + v'_{62}g) \gg 62$
 - 10: $g'_{62} \leftarrow (r'_{62}f + s'_{62}g) \gg 62$
-

```

• int jump64divsteps2_s255(unsigned long long count, unsigned long long
delta, unsigned long long f[4], unsigned long long g[4], unsigned
long long H[4]);

```

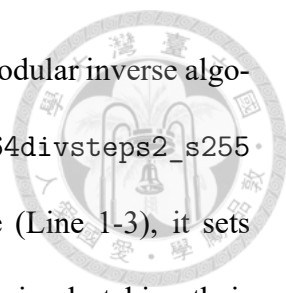
Input: $count = 62$; $\delta \in [-2^{60}, 2^{60}]$; $f, g \in [-2^{255}, 2^{255} - 1]$ where $2 \nmid f$;

Output: $(\delta'_{62}, f'_{62}, g'_{62}) = \text{divstep}^{62}(\delta, f, g)$; $H = \begin{bmatrix} u'_{62} & v'_{62} \\ r'_{62} & s'_{62} \end{bmatrix} = 2^{62}M_{62}(\delta, f, g)$

where

$\delta'_{62} \in [-60 - \delta, -60 + \delta]$; $f'_{62}, g'_{62} \in [-2^{255}, 2^{255} - 1]$;

$u'_{62}, v'_{62}, r'_{62}, s'_{62} \in [-2^{62} + 1, 2^{62}]$;



This is the most important part of computation to compose the modular inverse algorithm. This is also the most problematic subroutine to verify. `jump64divsteps2_s255` is the subroutine that computes 62 divstep iterations. To initialize (Line 1-3), it sets (u, v, r, s) to $(1, 0, 0, 1)$ and truncates full precision f, g into \bar{f}, \bar{g} by simply taking their lower 64 bits and treat them as signed 64-bit. The loop part (Line 4-6) is a loop that iterates 31 times where each iteration computes 2 divstep iterations. The last part (Line 7-10) computes the output. δ'_{62} and $(u'_{62}, v'_{62}, r'_{62}, s'_{62})$ are the results from the loop. To compute full precision f'_{62}, g'_{62} , we need to compute signed multi-limb multiplication with addition and 62-bit-shift.

To verify this subroutine, we first break down this subroutine into smaller pieces to verify some intermediate properties. Then, some verification tricks are used to combine the results and finish the verification. This section gives details on how we verify this subroutine.

4.3.2 Verify 1 divstep iteration

We first take a look at the loop of the subroutine, Line 4-6. As mentioned in Section 2.2.2, we use the loop variables u, v, r, s to store $2^i M_i$. Starting with δ , truncated \bar{f}, \bar{g} , and $(u, v, r, s) = (1, 0, 0, 1)$ (the identity matrix) as the inputs for the loop, the loop computes divstep iterations via $2^i M_i$ to keep the intermediate values in registers as integers.

Two implementations The computation of 62 divstep iterations is implemented with a loop that iterates 31 times where each iteration computes divstep twice. The purpose is to utilize more registers to avoid swaps. The result will always be in the desired registers after each iteration. To verify this loop body, we split the loop body into two parts (Line 5

and Line 6). Both parts compute one divstep, and the difference is just the set of registers used. Then, we verify both parts individually to show that the behavior of both parts follows the definition of divstep in Section 2.2.2.



Verify by case Given $2 \nmid f$, we can equivalently define divstep as:

$$\text{divstep} : (\delta, f, g) \mapsto \begin{cases} (\delta + 1, f, \frac{g}{2}) & \text{if } 2|g \\ (\delta + 1, f, \frac{g+f}{2}) & \text{if } \delta \leq 0 \text{ and } 2 \nmid g \\ (-\delta + 1, g, \frac{g-f}{2}) & \text{if } \delta > 0 \text{ and } 2 \nmid g \end{cases}$$

The definition of divstep depends on the value of input parameters and can be split into 3 cases. We verify the program under these three different conditions individually. To verify each case, we first not only specify the precondition but also assume that the input parameters meet the desired case. Each of the results of one iteration is kept in a 64-bit register, so the results here have only 64-bit precision. We use BOOLECTOR to prove that under each of the 3 cases, the corresponding result will be correctly computed by the program.

Take one of the implementations of a divstep as an example (Line 5), $(\delta, \bar{f}, \bar{g}, u, v, r, s, m)$ is the set of input variables and $(\delta', \bar{f}', \bar{g}', u', v', r', s', m')$ is the set of output variables. Starting from 1, m is used as an auxiliary variable that is doubled every iteration to specify the output range of (u', v', r', s') .

The range property after i -th divstep iteration is shown in Listing 4.2. Notice that this range property in Listing 4.2 is a CNF. To assume this property in SINGULAR (except the properties related to m , since we don't need the range in SINGULAR), we can transform

it to a DNF and specify the DNF in SINGULAR using the trick in Section 3.3.2.



or[

(* case 1 *)

and[$\delta \leq 0 \pmod{64}$, eqmod $\bar{g} \equiv 1 \pmod{64} \pmod{2 \pmod{64}}$,

$$\bar{g}' \pmod{2 \pmod{64}} = \bar{f} + \bar{g},$$

$$\bar{f}' = \bar{f},$$

$$(\text{sext } \delta' \pmod{1}) = 1 \pmod{65} + (\text{sext } \delta \pmod{1}),$$

$$(\text{sext } u' \pmod{1}) = (\text{sext } u \pmod{1}) \pmod{2 \pmod{65}},$$

$$(\text{sext } v' \pmod{1}) = (\text{sext } v \pmod{1}) \pmod{2 \pmod{65}},$$

$$(\text{sext } r' \pmod{1}) = (\text{sext } r \pmod{1}) + (\text{sext } u \pmod{1}),$$

$$(\text{sext } s' \pmod{1}) = (\text{sext } s \pmod{1}) + (\text{sext } v \pmod{1}),$$

$$(-1) \pmod{64} * m' \leq u', u' \leq m',$$

$$(-1) \pmod{64} * m' \leq v', v' \leq m',$$

$$(-1) \pmod{64} * m' \leq r', r' \leq m',$$

$$(-1) \pmod{64} * m' \leq s', s' \leq m',$$

$$1 \pmod{64} < m', m' \leq (2^{**62}) \pmod{64},$$

$$m' = m \pmod{2 \pmod{64}}$$

],

(* case 2 *)

and[$\delta > 0 \pmod{64}$, eqmod $\bar{g} \equiv 1 \pmod{64} \pmod{2 \pmod{64}}$,

$$\bar{g}' \pmod{2 \pmod{64}} = \bar{g} - \bar{f},$$



$$\bar{f}' = \bar{g},$$

$$(\text{sext } \delta' 1) = 1@65 - (\text{sext } \delta 1),$$

$$(\text{sext } u' 1) = (\text{sext } r 1) * 2@65,$$

$$(\text{sext } v' 1) = (\text{sext } s 1) * 2@65,$$

$$(\text{sext } r' 1) = (\text{sext } r 1) - (\text{sext } u 1),$$

$$(\text{sext } s' 1) = (\text{sext } s 1) - (\text{sext } v 1),$$

$$(-1)@64 * m' <=s u', u' <=s m',$$

$$(-1)@64 * m' <=s v', v' <=s m',$$

$$(-1)@64 * m' <=s r', r' <=s m',$$

$$(-1)@64 * m' <=s s', s' <=s m',$$

$$1@64 <s m', m' <=s (2**62)@64,$$

$$m' = m * 2@64$$

],

(* case 3 *)

and[eqmod \bar{g} 0@64 2@64,

$$(\text{sext } \bar{g}' 1) * 2@65 = (\text{sext } \bar{g} 1),$$

$$\bar{f}' = \bar{f},$$

$$(\text{sext } \delta' 1) = 1@65 + (\text{sext } \delta 1),$$

$$(\text{sext } u' 1) = (\text{sext } u 1) * 2@65,$$

$$(\text{sext } v' 1) = (\text{sext } v 1) * 2@65,$$

$$r' = r,$$

$$s' = s,$$



```

(-1)@64 * m' <=s u', u' <=s m',
(-1)@64 * m' <=s v', v' <=s m',
(-1)@64 * m' <=s r', r' <=s m',
(-1)@64 * m' <=s s', s' <=s m',
1@64 <s m', m' <=s (2**62)@64,
m' = m * 2@64
]
]

```

Listing 4.2: Range property after the $\log_2 m'$ -th divstep iteration.

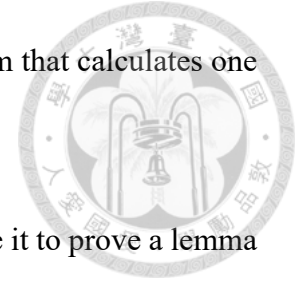
Nonetheless, when the results proved by BOOLECTOR have only 64-bit precision, it is not so easy to interpret these properties in SINGULAR, since properties are represented as mathematical equations in SINGULAR. To deal with this problem, we use the following trick:

Take one of the terms as an example: $\bar{g}' * 2@64 = \bar{g} - \bar{f}$ is proved by BOOLECTOR as one of the properties, where the variables $\bar{g}', \bar{g}, \bar{f}$ are 64 bit vectors. This means the MSB of (\bar{g}') is not used and BOOLECTOR does not check whether the subtraction causes overflow, i.e., the result has only 64-bit precision. This is equivalent to $\bar{g}' \times 2 \equiv \bar{g} - \bar{f} \pmod{2^{64}}$.

To express this property in SINGULAR as a mathematical equation, we accomplish this by introducing a free variable z . With z , we can precisely express this property as an equation:

$$(\bar{g}' * 2) - (\bar{g} - \bar{f}) - (z * 2^{64}) = 0$$

With these methods, we can express the behavior of the program that calculates one divstep iteration in SINGULAR.



After assuming the property of one divstep in SINGULAR, we use it to prove a lemma that BOOLECTOR cannot prove.

Remember that $\overset{\circ}{f}$ and $\overset{\circ}{g}$ denote the truncated f and g , we can use SINGULAR to prove the following lemma after the i -th divstep iteration:

Lemma 1. $u'\overset{\circ}{f} + v'\overset{\circ}{g} \equiv 0 \pmod{2^i}$, $r'\overset{\circ}{f} + s'\overset{\circ}{g} \equiv 0 \pmod{2^i}$, $u'\overset{\circ}{f} + v'\overset{\circ}{g} \equiv 2^i \bar{f}' \pmod{2^{64}}$, and $r'\overset{\circ}{f} + s'\overset{\circ}{g} \equiv 2^i \bar{g}' \pmod{2^{64}}$ after the i -th iteration, where $\overset{\circ}{f} = \text{truncate}(f, 64)$, $\overset{\circ}{g} = \text{truncate}(g, 64)$

This lemma can be directly proved by SINGULAR from the properties of a divstep that is assumed as a DNF. With the range properties and this lemma, we finished proving all the properties needed for one divstep iteration.

4.3.3 Model the subroutine

After verifying one divstep iteration separately, we can model the loop by using `call` as an abstraction layer in the main function of the subroutine with the techniques mentioned in Section 3.2.3. This way we can directly use the proved properties of one divstep without proving the same thing 62 times.

In this case, the function is one divstep iteration. For the precondition of the function, we use `assert` to check that the precondition of one divstep holds before every `call`. Then, we use `nondet` to initialize every output variable to avoid conflicts in the function body. Finally, we use `assume` to specify the properties of the output variables.

Since the implementation uses two different sets of registers to compute divstep^{62} , we alternate the parameters of the 62 calls according to the implementation.

After the 62 calls, we want to summarize the range of the output variables of the loop. While most of the properties can be extracted directly with an assert, there is one range property that BOOLECTOR cannot directly prove. However, we noticed that BOOLECTOR can prove a stronger set of properties which is the premise of the properties we want.

Denote (u', v', r', s') after the 62-th divstep iteration as $(u'_{62}, v'_{62}, r'_{62}, s'_{62})$. The following is the set of properties we used BOOLECTOR to prove:

$$-2^{62} < u'_{62}, v'_{62} \leq 2^{62}; |u'_{62}| + |v'_{62}| \leq 2^{62}; -2^{62} < u'_{62} + v'_{62}; -2^{255} \leq f, g < 2^{255} \quad (4.1)$$

And the range property we want is:

$$-2^{317} \leq u'_{62} * f + v'_{62} * g < 2^{317} \quad (4.2)$$

To make sure that Property 4.1 is indeed the premise of Property 4.2, we used Coq to prove this relation.

The verification of assembly implementations is a task in which CRYPTOLINE demonstrates utility, but its capabilities are limited when it comes to the verification of complex mathematical theorems. On the other hand, Coq is an open-source interactive theorem prover and formal verification tool that is useful for proving mathematical theorems. It

allows the user to write mathematical statements and proofs in formal language and provides a powerful proof assistant that verifies the correctness of these proofs using a type theory-based logic.



With the proof from Coq, we are confident that the properties we want will hold. Therefore, we directly assume Property 4.2 right after the assertion of Property 4.1 in CRYPTOLINE. This way we can use Property 4.2 to continue the verification process.

Before moving on to the last part of the computation, we use a cut to split the verification process into two parts to simplify the verification process. In the cut statement, we explicitly specify all the important properties of the results of the loop.

The properties in the cut statement contains:

1. The range of δ'_{62} is $[-60 - \delta, -60 + \delta]$,
2. The range of $u'_{62}, v'_{62}, r'_{62}, s'_{62}$ is $[-2^{62} + 1, 2^{62}]$,
3. Lemma 1 with $i = 62$ which shows that $u'_{62}f + v'_{62}g \equiv r'_{62}f + s'_{62}g \equiv 0 \pmod{2^{62}}$ and that they represent the desired output of 62 divstep iterations f'_{62}, g'_{62} ,
4. The range of $u'_{62}f + v'_{62}g$ and $r'_{62}f + s'_{62}g$ which is proved by Coq.

Afterwards, we can use this as the precondition for the rest of the program after the cut.

4.3.4 Verify signed multi-limb multiplication and shift

Finally, let's move on to the last part of the subroutine (Line 9-10). The last part computes the exact output of 62 divstep iterations with full 256-bit precision using the

results of the loop. This computation contains 256-to-64-bit signed multiplication and 62-bit-shift-to-right to compute:

$$f'_{62} = (u'_{62} * f + v'_{62} * g) \gg 62$$

$$g'_{62} = (r'_{62} * f + s'_{62} * g) \gg 62$$

With the tricks mentioned in Section 3.3.2, we also managed to verify this computation.

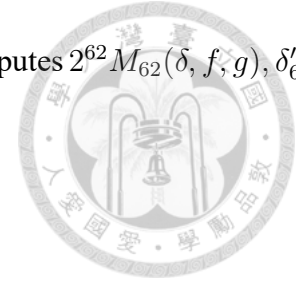
4.3.5 Completeness of verification of the subroutine

To state that we finished verifying this subroutine, we combine the following results:

1. In Subsection 4.3.2, we show that the computation of one divstep iteration (Line 4-5) is verified with 64-bit precision.
2. By Theorem 1 in Appendix A.1, we prove that the precision of the loop variables is sufficient. Therefore, the values of signed 64-bit (u', v', r', s') from the i -th iteration is exactly $2^i M_i$, and the 64-bit \bar{f}', \bar{g}' is sufficient to compute the next divstep iteration correctly throughout the loop. Combine this with the results in Subsection 4.3.2, we conclude that the results $(\delta'_{62}, u'_{62}, v'_{62}, r'_{62}, s'_{62})$ of 62 divstep iteration (Line 4-6) is verified with full precision.
3. In Subsection 4.3.3, we model the subroutine `jump64divsteps2_s255` and verified the range of $\delta'_{62}, u'_{62}, v'_{62}, r'_{62}, s'_{62}$ (Line 7-8).
4. In Subsection 4.3.3 and 4.3.4, we use Coq to verify the range of f'_{62}, g'_{62} and use CRYPTOLINE to verified that f'_{62}, g'_{62} are correctly computed using the output from the loop (Line 9-10).



With these results, we can state that this subroutine correctly computes $2^{62}M_{62}(\delta, f, g), \delta'_{62}, f'_{62}, g'_{62}$ and thus is verified.



4.4 Results

With all the subroutines verified, we are very close to claiming that the C implementation `fpinv25519.c` is also formally verified since all of the computations happen in these subroutines. The remaining question is whether we called these subroutines correctly. However, this problem is very tricky because this involves proving some highly non-trivial mathematical theorems in [1]², which seems to be not easily provable by solvers/tools of the time. We may just use the proof in [1]² for the correctness of the C implementation. To formally verify the theorems², we leave it as future work.

²Such as Theorem 11.2 in [1] with $f_0 = 2^{255} - 19$, which shows 744 iterations are indeed sufficient to compute the modular inverse.





Chapter 5 Verifying a Fast Vectorized Implementation

With the introduction of the improved Bernstein-Yang algorithm in Section 2.3, less number of divstep iterations are needed to compute modular inversion. Instead of 744, we only need to compute 600 divstep iterations when $f_0 = 2^{255} - 19$. Therefore, it is pretty clear that an implementation based on the improved Bernstein-Yang algorithm would have better performance.

In this chapter, we show another case study of the verification of a fast vectorized implementation of the improved Bernstein-Yang algorithm. Section 5.1 describes a fast vectorized implementation of the improved Bernstein-Yang algorithm. From Section 5.2 to 5.6, we describe the details of the computation and explain how to verify them. Section 5.7 explains the verification result.

The implementation and the verification code can be found at <https://github.com/fmlab-iis/cryptoline/tree/master/examples>.



5.1 Vectorized x86 25519 Implementation

This vectorized x86 implementation with $f_0 = 2^{255} - 19$ is also written by Bo-Yin Yang. It is an assembly implementation of the improved Bernstein-Yang algorithm that utilizes both SISD and SIMD on x86 with some delicate implementation choices. It takes only 3880 cycles on Intel Skylake core, which makes it the fastest implementation compare to the state-of-the-art.

Again, we follow the notation in Chapter 2. We use the notation

$$(\delta_i, f_i, g_i) = \text{divstep}^i(\delta_0, f_0, g_0)$$

and

$$\begin{bmatrix} u_i & v_i \\ r_i & s_i \end{bmatrix} = M_i(\delta_0, f_0, g_0)$$

in this chapter.

Similar to the implementation in Chapter 4, by theorem, we can get the modular inverse by computing $g_0^{-1} = f_{600}v_{600} \pmod{f_0}$ where

$$(\delta_{600}, f_{600}, g_{600}) = \text{divstep}^{600}\left(\frac{1}{2}, f_0, g_0\right); f_{600} \in \{1, -1\}$$

and

$$\begin{bmatrix} u_{600} & v_{600} \\ r_{600} & s_{600} \end{bmatrix} = M_{600}\left(\frac{1}{2}, f_0, g_0\right).$$

Again, we can say that the goal of this algorithm is to compute $f_{600}v_{600} \pmod{f_0}$.

In the following sections, we will describe the implementation choices to exploit the

potential of assembly programming.



5.1.1 Outline of the assembly code

Pseudo code We provide the pseudo code of this implementation in Algorithm 2. We use \bar{x} when it represents a truncated value of x that is in a lower precision. Also, we use capital letters F, V, G, S to denote radix 2^{30} numbers in full precision where taking F as example, $F = \sum_{i=0}^8 2^{30i} F[i]$ and $F[i]$ are 64-bit variables. Moreover, we use $\tilde{f}[0], \tilde{f}[1]$ to denote the first two coefficients of F as a radix 2^{60} number where $\tilde{f}[0] + 2^{60} \tilde{f}[1] = F \bmod 2^{120}$.

The program is written in assembly. All of the computations are written in one assembly subroutine. To verify this program, we split the program into small code blocks by their functionality for simplicity. Notice that there is no overhead caused by calling convention or maintaining the stack since they belong to the same subroutine. For consistency, we still call these code blocks "subroutines" in this chapter.

This outline of the program is depicted as follows:

1. Initialization (Line 1-7):

Given input $g_0 \in [0, 2^{256} - 1]$, the program reduces g_0 to $\mathbb{Z}_{2^{255}-19}$ (Line 2) and represents it in a radix- 2^{30} number (Line 3). Then, it loads some useful constants from the memory to the stack (Line 4) and initializes $m = \delta_0 - \frac{1}{2}$ and the loop variables (Line 5-7).

2. 60 divsteps (Line 14-31):



Algorithm 2 Pseudo 25519 vectorized safegcd (600 iterations)

Input: $\delta_0 = \frac{1}{2}$, $f_0 = 2^{255} - 19$. $g_0 \in [0, 2^{256} - 1]$.

Output: $g_0^{-1} \in \mathbb{Z}_{2^{255}-19}$ when $\gcd\{2^{255}-19, g_0\} = 1$.

- 1: \triangleright Initialization:
 - 2: $x \leftarrow \text{mod25519}(g_0)$
 - 3: $G \leftarrow \text{split9}(x)$
 - 4: $(F, V, G, S) \leftarrow (2^{255} - 19, 0, G, \frac{1}{2}^{60}) \in \mathbb{Z}_{2^{255}-19}$
 - 5: $m \leftarrow 0$
 - 6: $(\tilde{f}[0], \tilde{g}[0], \tilde{f}[1], \tilde{g}[1]) \leftarrow (-19, G[0] + 2^{30}G[1], 0, 0)$
 - 7: $(\hat{u}, \hat{v}, \hat{r}, \hat{s}) \leftarrow (2^{60}, 0, 0, 2^{60})$
 - 8: \triangleright Bigloop:
 - 9: **for** $j \leftarrow 1$ to 10 **do**
 - 10: $(\bar{f}, \bar{g}) \leftarrow \text{transition_portion}(\tilde{f}[0], \tilde{g}[0], \tilde{f}[1], \tilde{g}[1], \hat{u}, \hat{v}, \hat{r}, \hat{s})$
 - 11: $(F, V, G, S) \leftarrow \text{vector_mul_30}(F, V, G, S, \hat{u}, \hat{v}, \hat{r}, \hat{s})$
 - 12: $(\tilde{f}[0], \tilde{f}[1]) \leftarrow (F[0] + 2^{30}F[1], F[2] + 2^{30}F[3])$
 - 13: $(\tilde{g}[0], \tilde{g}[1]) \leftarrow (G[0] + 2^{30}G[1], G[2] + 2^{30}G[3])$
 - 14: \triangleright 60 divsteps:
 - 15: $(\bar{m}, \bar{f}, \bar{g}) \leftarrow (m, \bar{f}, \bar{g})$ \triangleright Saved for verification purpose.
 - 16: $(\hat{u}, \hat{v}, \hat{r}, \hat{s}) \leftarrow (1, 0, 0, 1)$
 - 17: **for** $i \leftarrow 1$ to 2 **do**
 - 18: $(\bar{f}, \bar{g}) = (\bar{f} \bmod 2^{20}, \bar{g} \bmod 2^{20})$
 - 19: $(u, v, r, s) \leftarrow (-1, 0, 0, -1)$
 - 20: $fwx \leftarrow \bar{f} + 2^{41}u + 2^{62}v$
 - 21: $gyz \leftarrow \bar{g} + 2^{41}r + 2^{62}s$
 - 22: $(m, fwx, gyz) \leftarrow \text{loop20}(m, fwx, gyz)$
 - 23: $(u, v, r, s) \leftarrow \text{extract}(fwx, gyz)$
 - 24: $(\hat{u}, \hat{v}, \hat{r}, \hat{s}, \bar{f}, \bar{g}) \leftarrow \text{updateuvrs}(\hat{u}, \hat{v}, \hat{r}, \hat{s}, u, v, r, s, \bar{f}, \bar{g})$
 - 25: $(\bar{f}, \bar{g}) = (\bar{f} \bmod 2^{20}, \bar{g} \bmod 2^{20})$
 - 26: $(u, v, r, s) \leftarrow (-1, 0, 0, -1)$
 - 27: $fwx \leftarrow \bar{f} + 2^{41}u + 2^{62}v$
 - 28: $gyz \leftarrow \bar{g} + 2^{41}r + 2^{62}s$
 - 29: $(m, fwx, gyz) \leftarrow \text{loop20}(m, fwx, gyz)$
 - 30: $(u, v, r, s) \leftarrow \text{extract}(fwx, gyz)$
 - 31: $(\hat{u}, \hat{v}, \hat{r}, \hat{s}) \leftarrow \text{lastloop}(\hat{u}, \hat{v}, \hat{r}, \hat{s}, u, v, r, s)$
 - 32: \triangleright Termination:
 - 33: $(\check{u}, \check{v}) \leftarrow \text{lastuv}(\tilde{f}[0], \tilde{f}[1], \tilde{g}[0], \tilde{g}[1], \hat{u}, \hat{v})$
 - 34: $g_0^{-1} \leftarrow \text{cneg}(V, S, \check{u}, \check{v})$
-

Given input $(\mathring{m}, \mathring{f}, \mathring{g})$, this part will compute the output



$$(m + \frac{1}{2}, f_{dontcare}, g_{dontcare}) := \text{divstep}^{60}(m + \frac{1}{2}, \mathring{f}, \mathring{g})$$

and

$$\begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} := -2^{60} M_{60}(m + \frac{1}{2}, \mathring{f}, \mathring{g})$$

To compute 60 divstep iterations, it computes 20 divstep iterations with 21-bit precision three times. For the first and second time where $i = 1, 2$, it first reduces the precision of (\bar{f}, \bar{g}) to 20-bit $(\bar{\bar{f}}, \bar{\bar{g}})$ (Line 18)¹. Then, it computes (Line 19-23)

$$(m + \frac{1}{2}, f_{dontcare}, g_{dontcare}) := \text{divstep}^{20}(m + \frac{1}{2}, \bar{\bar{f}}, \bar{\bar{g}})$$

and

$$\begin{bmatrix} u & v \\ r & s \end{bmatrix} := -2^{20} M_{20}(m + \frac{1}{2}, \bar{\bar{f}}, \bar{\bar{g}}).$$

Lastly, it updates (Line 24)

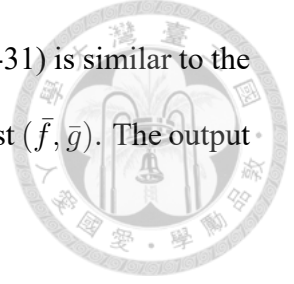
$$\begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} := \begin{bmatrix} u & v \\ r & s \end{bmatrix} \begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} = (-2^{20})^i M_{20i}(m + \frac{1}{2}, \mathring{f}, \mathring{g})$$

and

$$\begin{bmatrix} \bar{f} \\ \bar{g} \end{bmatrix} := \frac{1}{2^{20}} \begin{bmatrix} u & v \\ r & s \end{bmatrix} \begin{bmatrix} \bar{f} \\ \bar{g} \end{bmatrix} \equiv (-1)^i \begin{bmatrix} \text{truncate}(f_{60(j-1)+20i}, 60 - 20i) \\ \text{truncate}(g_{60(j-1)+20i}, 60 - 20i) \end{bmatrix} \pmod{2^{60-20i}}$$

using the results from the previous 20 divstep iterations.

¹By Theorem 1 in Appendix A.1, truncating to b -bits is equivalent to computing $(\text{mod } 2^b)$ because their results are equivalent in \mathbb{Z}_{2^b} .



As for the third 20 divstep iterations, the computation (Line 25-31) is similar to the computation of the first two except that it does not update the last (\bar{f}, \bar{g}) . The output of the third 20 divstep iterations will be

$$(m + \frac{1}{2}, f_{dontcare}, g_{dontcare}) := \text{divstep}^{60}(m + \frac{1}{2}, f, g)$$

and

$$\begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} := -2^{60} M_{60}(m + \frac{1}{2}, f, g)$$

3. Bigloop (Line 8-31):

Bigloop iterates 10 times with $j = 1, 2, 3, \dots, 10$. Each Bigloop iteration computes 60 divstep iterations and maintains the global variables F, V, G, S in full precision.

At the beginning of a Bigloop iteration, it first computes (Line 10)

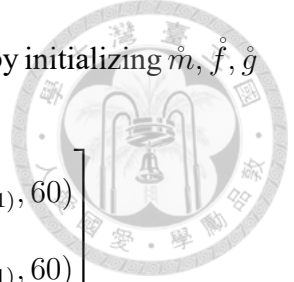
$$\begin{bmatrix} \bar{f} \\ \bar{g} \end{bmatrix} := (-1)^{j-1} \begin{bmatrix} \text{truncate}(f_{60(j-1)}, 60) \\ \text{truncate}(g_{60(j-1)}, 60) \end{bmatrix} \pmod{2^{60}}$$

using loop variables from the previous iteration. Then, it computes (Line 11)

$$\begin{bmatrix} F & V \\ G & S \end{bmatrix} := \frac{1}{2^{60}} \begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} \begin{bmatrix} F & V \\ G & S \end{bmatrix} = (-1)^{j-1} \begin{bmatrix} f_{60(j-1)} & \frac{1}{2^{60}} v_{60(j-1)} \\ g_{60(j-1)} & \frac{1}{2^{60}} s_{60(j-1)} \end{bmatrix} \pmod{2^{255}-19}$$

with full precision expressed as radix 2^{30} numbers and update $(\tilde{f}[0], \tilde{f}[1]), (\tilde{g}[0], \tilde{g}[1])$ with the computed F, G (Line 12-13).

Notice that in the first Bigloop iteration where $j = 1$, Line 10-13 are redundant and can be merged in the initialization, but we still keep them in the code for simplicity of execution.



Nextly, it prepares (Line 14-16) to compute 60 divstep iteration by initializing $\mathring{m}, \mathring{f}, \mathring{g}$

where

$$\mathring{m} = m_{60(j-1)}; \begin{bmatrix} \mathring{f} \\ \mathring{g} \end{bmatrix} = (-1)^{j-1} \begin{bmatrix} \text{truncate}(f_{60(j-1)}, 60) \\ \text{truncate}(g_{60(j-1)}, 60) \end{bmatrix}$$

and

$$\begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = 2^0 M_0(\mathring{m} + \frac{1}{2}, \mathring{f}, \mathring{g}).$$

With these variables, it starts to compute 60 divstep iterations on them.

4. Termination (Line 32-34):

After 10 Bigloop iterations, the program holds

$$\tilde{f}[0] + 2^{60}\tilde{f}[1] \equiv f_{540} \pmod{2^{120}}, \tilde{g}[0] + 2^{60}\tilde{g}[1] \equiv g_{540} \pmod{2^{120}},$$

$$\begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} = -2^{60} M_{60}(\delta_{540}, f_{540}, g_{540})$$

$$V \text{ and } S \text{ where } \begin{bmatrix} U & V \\ R & S \end{bmatrix} = -\frac{1}{2^{60}} M_{540}(\delta_0, f_0, g_0) \pmod{2^{255} - 19}.$$

With these results, it first computes² (Line 33) $f_{600} \in \{1, -1\}$ and set (\check{u}, \check{v}) as:

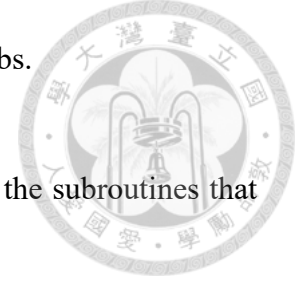
$$(\check{u}, \check{v}) := (f_{600}\hat{u}, f_{600}\hat{v}).$$

Finally, $g_0^{-1} \in \mathbb{Z}_{2^{255}-19}$ can be computed by (Line 34):

$$g_0^{-1} \equiv f_{600} * v_{600} \equiv \begin{bmatrix} f_{600}\hat{u} & f_{600}\hat{v} \end{bmatrix} \begin{bmatrix} V \\ S \end{bmatrix} = \check{u}V + \check{v}S \pmod{2^{255} - 19}$$

² $f_{600} = 1$ or -1 by theorem.

The result is expressed in binary representation in 4 64-bit limbs.



In Section 5.2, 5.3, 5.4, and 5.5, we will take a deeper look at the subroutines that compose this program while explaining how to verify them.

5.2 Verify 20 divstep iterations

- $(m_{20}, fwx_{20}, gyz_{20}) := \text{loop20}(m, fwx, gyz)$

Input:

$$m \in [-600, 600]; fwx = f + 2^{41}u + 2^{62}v; gyz = g + 2^{41}r + 2^{62}s$$

where

$$m = \delta - \frac{1}{2}; f, g \in [0, 2^{20} - 1]; \begin{bmatrix} u & v \\ r & s \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

Output:

$$m_{20}; fwx_{20} = \phi_{20} + 2^{21}w_{20} + 2^{42}x_{20}; gyz_{20} = \psi_{20} + 2^{21}y_{20} + 2^{42}z_{20}$$

where

$$(m_{20} + \frac{1}{2}, \phi_{20}, \psi_{20}) = \text{divstep}^{20}(m + \frac{1}{2}, f, g);$$

$$\begin{bmatrix} w_{20} & x_{20} \\ y_{20} & z_{20} \end{bmatrix} = -2^{20}M_{20}(m + \frac{1}{2}, f, g) \text{ and } w_{20}, x_{20}, y_{20}, z_{20} \in [-2^{20}, 2^{20} - 1].$$

Let's take a look at the subroutine that computes 20 divstep iterations first. `loop20` is the inner loop in `Bigloop` that loops three times and each computes 20 divstep iterations (Line 22 and Line 29 in Algorithm 2). Different from the Original Bernstein-Yang algorithm, the improved algorithm starts with $\delta_0 = \frac{1}{2}$, so the loop variable δ is always a fraction. To compute this version, some different implementation choices are applied.

loop20 computes $\text{divstep}^k(\delta, f, g)$ at its k -th iteration. In this section, with input (m, f, g) and $\delta = m + \frac{1}{2}$, we define

$$(d_0, m_0, \phi_0, \psi_0, w_0, x_0, y_0, z_0) = (\delta, m, f, g, u, v, r, s)$$

and

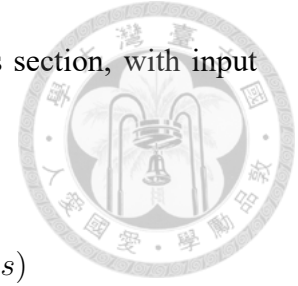
$$(d_k, \phi_k, \psi_k) = \text{divstep}^k(\delta, f, g); m_k = d_k - \frac{1}{2}; \begin{bmatrix} w_k & x_k \\ y_k & z_k \end{bmatrix} = -2^k M_k(\delta, f, g)$$

where $k = 1, 2, 3, \dots, 20$.

5.2.1 An alternative definition of divstep

In this implementation, to utilize the registers as integers, we use a signed 64-bit register m to store the value $\delta - \frac{1}{2}$ which is an integer. Also, to make the best use of the range of signed bit vectors, instead of calculating $2^k M_k$, we calculate $-2^k M_k$ in the k -th iteration. This is because the range of the entries of $2^k M_k(\delta, f, g)$ after the k -th divstep iteration lies in $[-2^k + 1, 2^k]$ which requires a signed $(k + 2)$ -bit vector to store. If we calculate $-2^k M_k(\delta, f, g)$ instead, the range will become $[-2^k, 2^k - 1]$ which can be stored in signed $(k + 1)$ -bit vectors.

With these implementation choices, we can equivalently define divstep with $m = \delta - \frac{1}{2}$ as follow:





$$\text{divstep}_m : (m, \phi, \psi) \mapsto \begin{cases} (m+1, \phi, \frac{\psi}{2}) & \text{if } 2|\psi \\ (m+1, \phi, \frac{\psi+\phi}{2}) & \text{if } m < 0 \text{ and } 2 \nmid \psi \\ (-m, \psi, \frac{\psi-\phi}{2}) & \text{if } m \geq 0 \text{ and } 2 \nmid \psi \end{cases}$$

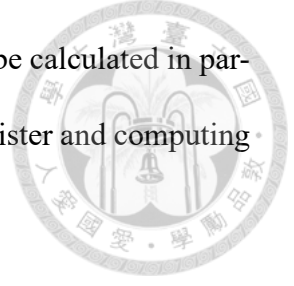
Notice that we can still apply the definitions in Section 2.2.1 and 2.2.2, but in this implementation, we compute with $m = \delta - \frac{1}{2}$ and $-2^k M_k$ instead.

5.2.2 Verify each divstep iteration

loop20 iterates 20 times. Each iteration calculates one divstep iteration using the loop variables $(m_k, \phi_k, \psi_k, w_k, x_k, y_k, z_k)$. At the start of the loop, it initializes $(w_0, x_0, y_0, z_0) := (u, v, r, s) = (-1, 0, 0, -1)$. We can view (w_0, x_0, y_0, z_0) as the negative 2×2 identity matrix in the initialization of loop20.

By definition, the k -th divstep iteration can be computed using the matrix representation, $k = 1, 2, \dots, 20$:

$$(m_k, \begin{bmatrix} \phi_k & w_k & x_k \\ \psi_k & y_k & z_k \end{bmatrix}) := \begin{cases} \left(m_{k-1} + 1, \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} \frac{\phi_{k-1}}{2} & w_{k-1} & x_{k-1} \\ \frac{\psi_{k-1}}{2} & y_{k-1} & z_{k-1} \end{bmatrix} \right) & \text{if } 2|\psi_{k-1} \\ \left(m_{k-1} + 1, \begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} \frac{\phi_{k-1}}{2} & w_{k-1} & x_{k-1} \\ \frac{\psi_{k-1}}{2} & y_{k-1} & z_{k-1} \end{bmatrix} \right) & \text{if } m_{k-1} < 0 \text{ and } 2 \nmid \psi_{k-1} \\ \left(-m_{k-1}, \begin{bmatrix} 0 & 2 \\ -1 & 1 \end{bmatrix} \times \begin{bmatrix} \frac{\phi_{k-1}}{2} & w_{k-1} & x_{k-1} \\ \frac{\psi_{k-1}}{2} & y_{k-1} & z_{k-1} \end{bmatrix} \right) & \text{if } m_{k-1} \geq 0 \text{ and } 2 \nmid \psi_{k-1} \end{cases}$$



Notice that the computation of (ϕ_k, w_k, x_k) and (ψ_k, y_k, z_k) can be calculated in parallel. To be specific, Instead of assigning each variable to a 64-bit register and computing using the above definition, for $k = 0, 1, 2, \dots, 20$, we define

$$\begin{aligned} fwx_k &= (\text{sext } \phi_k \ 44) + (\text{sext } w_k \ 43) * 2^{40-k} + (\text{sext } x_k \ 62-k) * 2^{61-k} \\ gyz_k &= (\text{sext } \psi_k \ 44) + (\text{sext } y_k \ 43) * 2^{40-k} + (\text{sext } z_k \ 62-k) * 2^{61-k} \end{aligned} \quad (5.1)$$

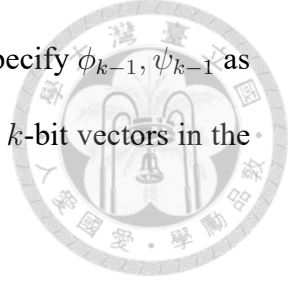
With this definition, we can just use three 64-bit variables m, fwx, gyz to store m_k, fwx_k, gyz_k where the content of fwx_k, gyz_k are depicted as follow:

| fwx | | | gyz | | |
|---------------|------------------|----------|---------------|------------------|----------|
| | sign of ϕ_k | ϕ_k | | sign of ψ_k | ψ_k |
| sign of w_k | w_k | | sign of y_k | y_k | |
| x_k | | | z_k | | |
| $k + 3$ | 21 | $40 - k$ | $k + 3$ | 21 | $40 - k$ |

Let $(fwx_0, gyz_0) = (fwx, gyz)$ be the input of the first iteration. With this definition, we can calculate divstep parallelly where (fwx_k, gyz_k) is the output of the k -th iteration as:

$$(m_k, fwx_k, gyz_k) := \begin{cases} (m_{k-1} + 1, fwx_{k-1}, \frac{gyz_{k-1}}{2}) & \text{if } 2 | gyz_{k-1} \\ (m_{k-1} + 1, fwx_{k-1}, \frac{gyz_{k-1} + fwx_{k-1}}{2}) & \text{if } m_{k-1} < 0 \text{ and } 2 \nmid gyz_{k-1} \\ (-m_{k-1}, gyz_{k-1}, \frac{gyz_{k-1} - fwx_{k-1}}{2}) & \text{if } m_{k-1} \geq 0 \text{ and } 2 \nmid gyz_{k-1} \end{cases}$$

This is exactly what one loop20 iteration computes. To verify this computation,



besides m_{k-1} , fwx_{k-1} , gyz_{k-1} as 64-bit vectors, we also explicitly specify ϕ_{k-1} , ψ_{k-1} as 20-bit vectors, w_{k-1} , y_{k-1} as signed 21-bit vectors, and x_{k-1} , z_{k-1} as k -bit vectors in the input parameters of each divstep. Then, we specify the range of

$$(m_{k-1}, \phi_{k-1}, \psi_{k-1}, w_{k-1}, x_{k-1}, y_{k-1}, z_{k-1})$$

and the relation between fwx_{k-1} , gyz_{k-1} and ϕ_{k-1} , ψ_{k-1} , w_{k-1} , x_{k-1} , y_{k-1} , z_{k-1} as the definition in Property 5.1.

Notice that the range of w_{k-1} , x_{k-1} , y_{k-1} , z_{k-1} lies in $[-2^{k-1}, 2^{k-1} - 1]$ in the precondition which will be verified in the previous iteration.

Secondly, we use some simple modeling techniques to model the computation instructions in CRYPTO_{LINE}. Thirdly, after the program body, we manually extract the signed 20-bit ϕ_k , ψ_k , signed 21-bit w_k , y_k , and signed $(k + 1)$ -bit x_k , z_k with some additional CRYPTO_{LINE} instructions. Finally, in the postcondition, we state the properties of

$$(m_k, \phi_k, \psi_k, w_k, x_k, y_k, z_k)$$

in the three cases as a CNF, similar to the properties in Listing 4.2 in Section 4.3.2. In addition, we also state the relation of fwx_k , gyz_k and their represented variables.

Notice that the range of w_k , x_k , y_k , z_k lies in $[-2^k, 2^k - 1]$ which is verified by CRYPTO_{LINE} in the postcondition, so the bit width for them in `fwx` and `gyz` are enough to keep their values.

We verified the postcondition of each of the 20 iterations separately and make sure that the postcondition of an iteration is exactly the precondition of the next iteration. After

CRYPTOLINE checked all of the 20 iterations, we can conclude that the 20 iterations of loop20 will compute



$$(m_{20}, fw_{20}, gyz_{20}, \phi_{20}, \psi_{20}, w_{20}, x_{20}, y_{20}, z_{20})$$

, and by definition, this will be the desired result after 20 divstep iterations where

$$(m_{20} + \frac{1}{2}, \phi_{20}, \psi_{20}) = \text{divstep}^{20}(m + \frac{1}{2}, f, g)$$

and

$$\begin{bmatrix} w_{20} & x_{20} \\ y_{20} & z_{20} \end{bmatrix} = -2^{20} M_{20}(m + \frac{1}{2}, f, g).$$

We verified the outputs fit their precision and they will not go out of their ranges. Moreover, we use SINGULAR to prove a lemma similar to the one in Section 4.3.2, but with the variables $(m_k, \phi_k, \psi_k, w_k, x_k, y_k, z_k)$. Hence, the loop20 implementation is verified.

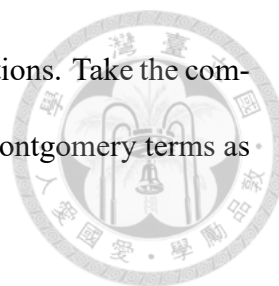
5.3 Verify vectorized update

5.3.1 Pseudo code of the subroutine

Pseudo code To explain the computation in this subroutine, we provide the pseudo code of the subroutine `vec_mul_30` in Algorithm 3.

This Algorithm works as follows:

- (Line 1-3) For each input u, v, r, s , express it as a radix 2^{30} number and keep each of them in two 64-bit registers.



- (Line 4-27) Compute four independent Montgomery multiplications. Take the computation of V' in Lane 2 as an example, it first computes the Montgomery terms as radix 2^{30} numbers by (Line 4-8)

$$d_V[0] + 2^{30}d_V[1] = d_V = ((-(2^{255} - 19)^{-1} \bmod 2^{60}) * V) \bmod 2^{60}.$$

Then, it computes (Line 9-27)

$$V' = uV + rS + d_V(2^{255} - 19)$$

- (Line 28-32) Again, we take the computation of V' in Lane 2 as an example, it computes

$$V' := \frac{1}{2^{60}}V'$$

and reduce it to its desired range.



Algorithm 3 Pseudo `vec_mul_30`

Input: $u, v, r, s \in [-2^{60}, 2^{60}]$, $F, G \in [-2^{255} + 1, 2^{255} - 1]$, $V, S \in [-2^{29} + 1, 2^{255} + 2^{29} - 1]$.

Output: $F', G' \in [-2^{255} + 1, 2^{255} - 1]$, $V', S' \in [-2^{29} + 1, 2^{255} + 2^{29} - 1]$.

- 1: \triangleright Transform to radix 2^{30} :
 - 2: $(u[0], v[0], r[0], s[0]) \leftarrow (u \bmod 2^{30}, v \bmod 2^{30}, r \bmod 2^{30}, s \bmod 2^{30})$
 - 3: $(u[1], v[1], r[1], s[1]) \leftarrow (u \gg 30, v \gg 30, r \gg 30, s \gg 30)$
 - 4: \triangleright Montgomery term:
 - 5: $(d_F, d_G, d_V, d_S) \leftarrow \text{MontgomeryTerm}(u[0], v[0], r[0], s[0], u[1], v[1], r[1], s[1], F, V, G, S)$
 - 6: \triangleright Transform to radix 2^{30} :
 - 7: $(d_F[0], d_G[0], d_V[0], d_S[0]) \leftarrow (d_F \bmod 2^{30}, d_G \bmod 2^{30}, d_V \bmod 2^{30}, d_S \bmod 2^{30})$
 - 8: $(d_F[1], d_G[1], d_V[1], d_S[1]) \leftarrow (d_F \gg 30, d_G \gg 30, d_V \gg 30, d_S \gg 30)$
 - 9: \triangleright radix 2^{30} multi-limb multiplication with addition:
 - 10: $\text{carry}_F, F'[0] \leftarrow F[0]u[0] + G[0]v[0]$
 - 11: $\text{carry}_V, V'[0] \leftarrow V[0]u[0] + S[0]v[0]$
 - 12: $\text{carry}_G, G'[0] \leftarrow F[0]r[0] + G[0]s[0]$
 - 13: $\text{carry}_S, S'[0] \leftarrow V[0]r[0] + S[0]s[0]$
 - 14: **for** $i \leftarrow 1$ to 8 **do**
 - 15: $\text{carry}_F, F'[i] \leftarrow \text{carry}_F + F[i]u[0] + G[i]v[0] + F[i-1]u[1] + G[i-1]v[1]$
 - 16: $\text{carry}_V, V'[i] \leftarrow \text{carry}_V + V[i]u[0] + S[i]v[0] + V[i-1]u[1] + S[i-1]v[1]$
 - 17: $\text{carry}_G, G'[i] \leftarrow \text{carry}_G + F[i]r[0] + G[i]s[0] + F[i-1]r[1] + G[i-1]s[1]$
 - 18: $\text{carry}_S, S'[i] \leftarrow \text{carry}_S + V[i]r[0] + S[i]s[0] + V[i-1]r[1] + S[i-1]s[1]$
 - 19: $F'[10], F'[9] \leftarrow \text{carry}_F + F_8u[1] + G_8v[1]$
 - 20: $V'[10], V'[9] \leftarrow \text{carry}_V + V_8u[1] + S_8v[1]$
 - 21: $G'[10], G'[9] \leftarrow \text{carry}_G + F_8r[1] + G_8s[1]$
 - 22: $S'[10], S'[9] \leftarrow \text{carry}_S + V_8r[1] + S_8s[1]$
 - 23: \triangleright radix 2^{30} multi-limb multiplication-by-19 with addition:
 - 24: $F' \leftarrow F' + d_F[0](2^{255} - 19) + d_F[1](2^{255} - 19) \quad \triangleright F' \equiv 0 \pmod{2^{60}}$ at this point.
 - 25: $V' \leftarrow V' + d_V[0](2^{255} - 19) + d_V[1](2^{255} - 19) \quad \triangleright V' \equiv 0 \pmod{2^{60}}$ at this point.
 - 26: $G' \leftarrow G' + d_G[0](2^{255} - 19) + d_G[1](2^{255} - 19) \quad \triangleright G' \equiv 0 \pmod{2^{60}}$ at this point.
 - 27: $S' \leftarrow S' + d_S[0](2^{255} - 19) + d_S[1](2^{255} - 19) \quad \triangleright S' \equiv 0 \pmod{2^{60}}$ at this point.
 - 28: \triangleright shift and reduce range:
 - 29: $F' \leftarrow F' \gg 60 \in [-2^{255} + 1, 2^{255} - 1]$
 - 30: $V' \leftarrow V' \gg 60 \in [-2^{29} + 1, 2^{255} + 2^{29} - 1]$
 - 31: $G' \leftarrow G' \gg 60 \in [-2^{255} + 1, 2^{255} - 1]$
 - 32: $S' \leftarrow S' \gg 60 \in [-2^{29} + 1, 2^{255} + 2^{29} - 1]$
-



- $(F', V', G', S') := \text{vec_mul_30}(F, V, G, S, u, v, r, s)$

Input:

$u, v, r, s \in [-2^{60}, 2^{60}]$ in 64-bit registers; $F, G \in [-2^{255} + 1, 2^{255} - 1]$; and $V, S \in [-2^{29} + 1, 2^{255} + 2^{29} - 1]$. Each of F, V, G, S are kept in 9 in 64-bit locations

where

$$|u| + |v| \leq 2^{60}; |r| + |s| \leq 2^{60};$$

$$F = \sum_{i=0}^8 2^{30i} F[i]; F[i] \in [0, 2^{30} - 1] \text{ for } i = 0, 1, \dots, 7; F[8] \in [-2^{16} + 1, 2^{15} - 1];$$

$$V = \sum_{i=0}^8 2^{30i} V[i]; V[0] \in [-2^{29} + 1, 2^{30} + 2^{29} - 1]; V[i] \in [0, 2^{30} - 1] \text{ for } i = 1, 2, \dots, 7; V[8] \in [0, 2^{15} - 1];$$

$$G = \sum_{i=0}^8 2^{30i} G[i]; G[i] \in [0, 2^{30} - 1] \text{ for } i = 0, 1, \dots, 7; G[8] \in [-2^{16} + 1, 2^{15} - 1];$$

$$S = \sum_{i=0}^8 2^{30i} S[i]; S[0] \in [-2^{29} + 1, 2^{30} + 2^{29} - 1]; S[i] \in [0, 2^{30} - 1] \text{ for } i = 1, 2, \dots, 7; S[8] \in [0, 2^{15} - 1];$$

$$\text{and } uF + vG \equiv 0 \pmod{2^{60}}; rF + sG \equiv 0 \pmod{2^{60}}.$$

Output:

$$\begin{bmatrix} F' & V' \\ G' & S' \end{bmatrix} = \frac{1}{2^{60}} \begin{bmatrix} u & v \\ r & s \end{bmatrix} \begin{bmatrix} F & V \\ G & S \end{bmatrix} \pmod{2^{255} - 19}$$

Each of F', V', G', S' are kept in 9 64-bit locations

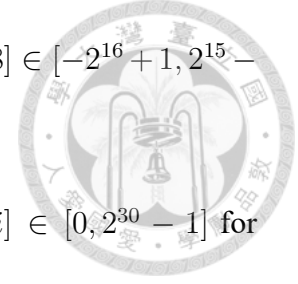
where

$$F' = \sum_{i=0}^8 2^{30i} F'[i]; F'[i] \in [0, 2^{30} - 1] \text{ for } i = 0, 1, \dots, 7; F'[8] \in [-2^{16} + 1, 2^{15} - 1];$$

$$V' = \sum_{i=0}^8 2^{30i} V'[i]; V'[0] \in [-2^{29} + 1, 2^{30} + 2^{29} - 1]; V'[i] \in [0, 2^{30} - 1] \text{ for } i = 1, 2, \dots, 7; V'[8] \in [0, 2^{15} - 1];$$

$G' = \sum_{i=0}^8 2^{30i} G'[i]; G'[i] \in [0, 2^{30} - 1]$ for $i = 0, 1, \dots, 7; G'[8] \in [-2^{16} + 1, 2^{15} - 1]$;

$S' = \sum_{i=0}^8 2^{30i} S'[i]; S'[0] \in [-2^{29} + 1, 2^{30} + 2^{29} - 1]; S'[i] \in [0, 2^{30} - 1]$ for $i = 1, 2, \dots, 7; S'[8] \in [0, 2^{15} - 1]$.



Let's take a look at a more complicated subroutine to verify, `vec_mul_30`. At the start of the 2nd to 10th `BigLoop` iteration (from $j = 2$ to $j = 10$), $\begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} = -2^{60} M_{60}(\delta_{60(j-2)}, f_{60(j-2)}, g_{60(j-2)})$ are computed with 64-bit precision (except for $j = 1$ this subroutine is idle). The algorithm needs to compute full precision $(-1)^{j-1} f_{60(j-1)}$, $(-1)^{j-1} g_{60(j-1)}$ and the right half of the full precision $(-1)^{j-1} \frac{1}{2^{60}} M_{60(j-1)}(\frac{1}{2}, 2^{255} - 19, g_0)$ and then keep them in the stack (Line 11 in Algorithm 2).

`vec_mul_30` is the subroutine for this computation. The full precision numbers F, V, G, S are represented as radix 2^{30} numbers and should be in $\mathbb{Z}_{2^{255}-19}$. However, the implementation does not reduce F, V, G, S to $\mathbb{Z}_{2^{255}-19}$ on the fly but will guarantee that every time they are put back into the stack they will stay in a fixed range. Notice that the range is the same for the inputs and outputs and will be verified by `CRYPTOLINE`.

5.3.2 Computing in parallel

Notice that the computation of F', V', G', S' can be computed in parallel using the SIMD instructions. The implementation uses `ymm` registers to compute. A 256-bit `ymm` register in x86 can be viewed as 4 64-bit lanes. Various AVX2 arithmetic instructions are supported to manipulate `ymm` registers and 256-bit memory locations.

It is not very difficult to model this implementation in `CRYPTOLINE`. A `ymm` register can be modeled by 4 64-bit `CRYPTOLINE` variables and a vector instruction can be modeled

by 4 CRYPTO LINE instructions. During verification, we will use CRYPTO LINE to check the computation in all 4 lanes independently.



5.3.3 Computing with Montgomery multiplication

Notice that the computation of the subroutine can be viewed as a modular matrix product:

$$\begin{bmatrix} F' & V' \\ G' & S' \end{bmatrix} := \frac{1}{2^{60}} \begin{bmatrix} u & v \\ r & s \end{bmatrix} \begin{bmatrix} F & V \\ G & S \end{bmatrix} \pmod{2^{255} - 19}$$

To compute multiplication in modular arithmetic, this implementation uses a technique called Montgomery multiplication [5]. It achieves fast computation by leveraging arithmetic shifts to reduce the number of multiplication instructions. To be specific, the implementation wants to compute $d_F, d_V, d_G, d_S \in [0, 2^{60} - 1]$ such that

$$(uF + vG + d_F(2^{255} - 19)) \equiv 0 \pmod{2^{60}}$$

$$(uV + vS + d_V(2^{255} - 19)) \equiv 0 \pmod{2^{60}}$$

$$(rF + sG + d_G(2^{255} - 19)) \equiv 0 \pmod{2^{60}}$$

$$(rV + sS + d_S(2^{255} - 19)) \equiv 0 \pmod{2^{60}}$$

Since F, V, G, S are represented as radix 2^{30} numbers with 9 coefficients, we naturally choose the base as 2^{30} to compute multi-limb Montgomery multiplication. First, u, v, r, s will be also transformed to radix 2^{30} numbers with 2 coefficients (Line 1-3). Second, we compute $d_F, d_V, d_G, d_S \in [0, 2^{60} - 1]$ by Montgomery multiplication and also express them as radix 2^{30} numbers with 2 coefficients (Line 4-8). We easily verified that

d_F, d_V, d_G, d_S satisfy their property with SINGULAR and BOOLECTOR.



Then, with d_F, d_V, d_G, d_S , the implementation computes (Line 9-27)

$$F := (uF + vG + d_F(2^{255} - 19)) \gg 60 \pmod{2^{255} - 19}$$

$$V := (uV + vS + d_V(2^{255} - 19)) \gg 60 \pmod{2^{255} - 19}$$

$$G := (rF + sG + d_G(2^{255} - 19)) \gg 60 \pmod{2^{255} - 19}$$

$$S := (rV + sS + d_S(2^{255} - 19)) \gg 60 \pmod{2^{255} - 19}$$

using radix 2^{30} multi-limb multiplication with addition.

To verify this computation, the verification engineer has to keep track of the sign of each coefficient of the radix 2^{30} numbers. Notice that as specified in the properties of the input and output, the sign of each coefficient of F, V, G, S is different where some are signed (for those that may be negative) and others are unsigned (for those in $[0, 2^{30} - 1]$).

`cast` and `vpc` are used to maintain well-formedness while keeping track of the values they represent. The type system in CRYPTOLINE captures the information of the sign of each coefficient and the safety check is passed to ensure that there shall be no overflow during the radix 2^{30} multi-limb multiplication with additions.

Notice that by the definition of `divstep`, it is guaranteed that $uF + vG \equiv 0 \pmod{2^{60}}$ and $rF + sG \equiv 0 \pmod{2^{60}}$. Therefore, $d_F = 0$ and $d_G = 0$ in this computation.



5.3.4 Verify signed shift right computed with unsigned shift right

Most of the computation of the radix 2^{30} multi-limb multiplication with additions can be verified with some simple CRYPTOLINE tricks. Here, we will present a special computation technique that is used and explain how to verify it.

During the computation, we need to compute signed-division-by- 2^{30} , namely, 30-bit-signed-shift-right on a signed 64-bit register in each lane. However, the platform only supports unsigned-shift-right for the vector instructions. To utilize vector instructions, we use unsigned-shift-right to simulate signed-shift-right. Specifically, for a signed 64-bit register `rax`, we compute

$$\text{rbx} := ((\text{rax} + 2^{63}) \gg_u 30) - 2^{63-30}$$

in 64-bit precision. Surprisingly, the result `rbx` will be equivalent to `rax` $\gg_s 30$ in 63-bit precision. Luckily, 63-bit precision is enough for this subroutine, so we can still compute with vector instructions.

To verify this computation, before we add/subtract the constants $2^{63}/2^{33}$, we cast `rax` to unsigned since adding/subtracting the constants might overflow. Also, we use `BOOLECTOR` to explicitly prove the properties after every shift and assume it in `SINGULAR` since shift which is a non-linear instruction. Lastly, we have to cast the result `rbx` back to signed for the computation later.

Listing 5.1 shows a simplified example. Notice that the properties of `rbx` that `BOOLECTOR` proved have only $93 - 30 = 63$ -bit precision. The verification engineer has to make

sure that only the lower 63 bits of `rbx` will be used in the rest of the program to assume this property in SINGULAR.



```
(* rax is sint64 *)
cast rbx@uint64 rax;
adds carry rbx rbx (2**(63))@uint64;
usplit rbx low rbx 30;
subb borrow rbx rbx (2**(63-30))@uint64;
cast rbx@sint64 rbx;
assert true &&
  (uext low 29) + (sext rbx 29) * (2**(30))@93 = (sext rax
    29);
assume low + rbx * (2**(30)) = rax && true;
(* rbx is the signed-shift-right-30 result of rax
  and only the lower 63 bits of rbx will be used *)
```

Listing 5.1: An example of simulating signed shift right using unsigned shift right.

5.3.5 Use a proof from Coq

Similar to the simple implementation in Chapter 4, there are some implications in the range property that BOOLECTOR cannot prove, so we turn to Coq again. In this case, we will use two kinds of implications from Coq. The first one is similar to the example in Section 4.3.3, the only difference is that we only compute 60 divstep iterations instead of 62, so the range of the inputs and outputs is slightly different. With the proof from Coq,

we can safely assume

$$-2^{315} < uF + vG < 2^{315}; -2^{315} < rF + sG < 2^{315}$$



in CRYPTO_{LINE} for BOOLECTOR.

The second implication is more complicated, let's take the computation of V' as an example (we can apply the same method for the computation of S'), remember that BOOLECTOR holds the following range property

$$|u| + |v| \leq 2^{60}; -2^{29} < V, S < 2^{255} + 2^{29}; 0 \leq d_V < 2^{60} \quad (5.2)$$

while the property we want is

$$-2^{316} < uV + vS + d_V * (2^{255} - 19) < 2^{316} \quad (5.3)$$

Again, we use Coq to check that Property 5.2 is indeed the premise of Property 5.3 formally. With the proof from Coq, we can assume Property 5.3 in BOOLECTOR can continue the verification.

5.3.6 Reduce the output range

Finally, the subroutine reduces the range of the output (Line 28-32). Notice that the 60-bit shift will be exact because of Montgomery multiplication, so this shift can be merged in the radix 2^{30} multi-limb multiplication with additions by simply manipulating the index of each coefficient.

With all these verification tricks, we can finally check all the properties in the post-condition which include the algebraic property



$$\begin{bmatrix} F' & V' \\ G' & S' \end{bmatrix} := \frac{1}{2^{60}} \begin{bmatrix} u & v \\ r & s \end{bmatrix} \begin{bmatrix} F & V \\ G & S \end{bmatrix} \pmod{2^{255} - 19}$$

and the range property

$$F' = \sum_{i=0}^8 2^{30i} F'[i]; F'[i] \in [0, 2^{30} - 1] \text{ for } i = 0, 1, \dots, 7; F'[8] \in [-2^{16} + 1, 2^{15} - 1];$$

$$V' = \sum_{i=0}^8 2^{30i} V'[i]; V'[0] \in [-2^{29} + 1, 2^{30} + 2^{29} - 1]; V'[i] \in [0, 2^{30} - 1] \text{ for } i = 1, 2, \dots, 7; V'[8] \in [0, 2^{15} - 1];$$

$$G' = \sum_{i=0}^8 2^{30i} G'[i]; G'[i] \in [0, 2^{30} - 1] \text{ for } i = 0, 1, \dots, 7; G'[8] \in [-2^{16} + 1, 2^{15} - 1];$$

$$S' = \sum_{i=0}^8 2^{30i} S'[i]; S'[0] \in [-2^{29} + 1, 2^{30} + 2^{29} - 1]; S'[i] \in [0, 2^{30} - 1] \text{ for } i = 1, 2, \dots, 7; S'[8] \in [0, 2^{15} - 1].$$

That being said, we verified the algebraic relation between the inputs and outputs and verified that the outputs will be in their range as specified. Hence, the subroutine `vec_mul_30` is verified.



5.4 Verify radix 2^{30} number multiplication with reduction

- $g_0^{-1} := \text{cneg}(V, S, \check{u}, \check{v})$

Input:

$\check{u}, \check{v} \in [-2^{60}, 2^{60}]$ in 64-bit registers and $V, S \in [-2^{29} + 1, 2^{255} + 2^{29} - 1]$. Each of V, S are kept in 9 64-bit locations

where

$$\check{u} = f_{600}\hat{u}, \check{v} = f_{600}\hat{v} \text{ and } \begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} = -2^{60}M_{60}(\delta_{540}, f_{540}, g_{540});$$

$$V = \sum_{i=0}^8 2^{30i}V[i]; V[0] \in [-2^{29} + 1, 2^{30} + 2^{29} - 1]; V[i] \in [0, 2^{30} - 1] \text{ for } i = 1, 2, \dots, 7; V[8] \in [0, 2^{15} - 1];$$

$$S = \sum_{i=0}^8 2^{30i}S[i]; S[0] \in [-2^{29} + 1, 2^{30} + 2^{29} - 1]; S[i] \in [0, 2^{30} - 1] \text{ for } i = 1, 2, \dots, 7; S[8] \in [0, 2^{15} - 1];$$

$$\text{and } \begin{bmatrix} U & V \\ R & S \end{bmatrix} \equiv -\frac{1}{2^{60}}M_{540}(\frac{1}{2}, 2^{255} - 19, g_0) \pmod{2^{255} - 19}.$$

Output:

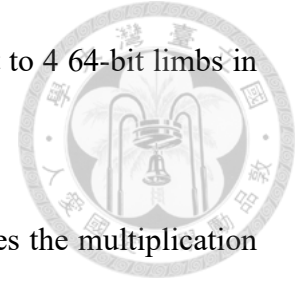
$$g_0^{-1} \in [0, 2^{255} - 19 - 1] \text{ represented as 4 64-bit unsigned limbs}$$

where

$$g_0^{-1} \equiv f_{600}v_{600} \equiv [f_{600}\hat{u} \quad f_{600}\hat{v}] \begin{bmatrix} V \\ S \end{bmatrix} \equiv \check{u}V + \check{v}S \pmod{2^{255} - 19}.$$

Let's take a look at another important subroutine in this implementation. `cneg` is the subroutine that is used in the last part of the computation of the algorithm (Line 33 in Algorithm 2). As we mentioned in Chapter 2.3, the modular inverse can be computed by $g_0^{-1} \equiv f_{600}v_{600} \pmod{f_0}$ where $\begin{bmatrix} u_{600} & v_{600} \\ r_{600} & s_{600} \end{bmatrix} = M_{600}(\delta_0, f_0, g_0)$. This is what exactly subroutine computes. Because we want the output of the implementation to be represented

as 4 64-bit limbs in $\mathbb{Z}_{2^{255}-19}$, this subroutine has to convert its result to 4 64-bit limbs in $\mathbb{Z}_{2^{255}-19}$.



This subroutine contains two parts. In the first part, it computes the multiplication of radix 2^{30} numbers (V, S) and 64-bit registers (\check{u}, \check{v}) with addition and represents the result as 5 64-bit signed limbs as a radix 2^{64} number. To compute this, it sequentially computes the multiplication of (\check{u}, \check{v}) and $(V[i], S[i])$ and adds the results to the 5 64-bit limbs destination. Because the radices of the sources and destination are different, the results are shifted before adding to the destination. This computation of the shift can be verified by BOOLECTOR, and consequently SINGULAR can show that the 5 64-bit signed limbs destination will be equal to $\check{u}V + \check{v}S$.

The second parts reduce the 5 64-bit signed limbs to $[0, 2^{255} - 19 - 1]$ which is computed by several multiplication-by-19 and additions. Notice that this involves computing sign extension. To verify the second part, we use BOOLECTOR to prove that the value stays the same after the sign extension and assume this in SINGULAR.

Finally, in the postcondition, we verified the algebraic property

$$g_0^{-1} \equiv \check{u}V + \check{v}S \pmod{2^{255} - 19}$$

with SINGULAR and the range property

$$g_0^{-1} \in [0, 2^{255} - 19 - 1]$$

with BOOLECTOR. Hence, the last subroutine `cneg` is also verified.



5.5 Verify simple subroutines

The program is made up of multiple subroutines. Finally, let's look at the simple ones. We will introduce them as the order of execution.

1. `mod25519`: Given $g_0 \in [0, 2^{256} - 1]$ in 4 64-bit limbs, compute $x \in \mathbb{Z}_{2^{255}-19}$ in 4 64-bit limbs such that $x \equiv g_0 \pmod{2^{255} - 19}$.
2. `split9`: Convert $x \in \mathbb{Z}_{2^{255}-19}$ in 4 64-bit limbs into a radix- 2^{30} number G . G is represented as $\sum_{i=0}^8 2^{30i} G[i]$ and is stored in 9 64-bit locations in the stack. Meanwhile, load some constants into the stack.
3. `transition_portion`: Given 64-bit precision $(\hat{u}, \hat{v}, \hat{r}, \hat{s})$ from the previous 60 divstep iterations, the 60-bit $(\tilde{f}[0], \tilde{f}[1], \tilde{g}[0], \tilde{g}[1])$ in $[0, 2^{60} - 1]$, compute 60-bit (\bar{f}, \bar{g}) for the current Bigloop iteration. The 60-bit output is computed by

$$\bar{f} := (\hat{u}(\tilde{f}[0] + 2^{60}\tilde{f}[1]) + \hat{v}(\tilde{g}[0] + 2^{60}\tilde{g}[1])) \gg 60$$

$$\bar{g} := (\hat{r}(\tilde{f}[0] + 2^{60}\tilde{f}[1]) + \hat{s}(\tilde{g}[0] + 2^{60}\tilde{g}[1])) \gg 60$$

Notice that the output satisfies

$$\begin{bmatrix} \bar{f} \\ \bar{g} \end{bmatrix} := (-1)^{j-1} \begin{bmatrix} \text{truncate}(f_{60(j-1)}, 60) \\ \text{truncate}(g_{60(j-1)}, 60) \end{bmatrix} \pmod{2^{60}}$$

in the j -th Bigloop iteration.

4. `extract and updatevrs`: After every first and second loop20 ($i = 1, 2$), extract



(u, v, r, s) from 64-bit fwx and gyz . Then, update $(\hat{u}, \hat{v}, \hat{r}, \hat{s})$ by

$$\begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} := \begin{bmatrix} u & v \\ r & s \end{bmatrix} \begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix}.$$

Meanwhile, update the 64-bit loop variables (\bar{f}, \bar{g}) for the next loop20 using the old $(60 - 20(i - 1))$ -bit precision (\bar{f}, \bar{g}) by

$$\begin{bmatrix} \bar{f} \\ \bar{g} \end{bmatrix} := \frac{1}{2^{20}} \begin{bmatrix} u & v \\ r & s \end{bmatrix} \begin{bmatrix} \bar{f} \\ \bar{g} \end{bmatrix}.$$

Notice that the output satisfies

$$\begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} = (-2^{20})^i M_{20i}(\delta_{60(j-1)}, f_{60(j-1)}, g_{60(j-1)})$$

and

$$\begin{bmatrix} \bar{f} \\ \bar{g} \end{bmatrix} \equiv (-1)^i \begin{bmatrix} \text{truncate}(f_{60(j-1)+20i}, 60 - 20i) \\ \text{truncate}(g_{60(j-1)+20i}, 60 - 20i) \end{bmatrix} \pmod{2^{60-20i}}$$

in the j -th Bigloop iteration.

5. **extract and lastloop:** After every third loop20, extract (u, v, r, s) from 64-bit fwx and gyz . Then, update $(\hat{u}, \hat{v}, \hat{r}, \hat{s})$ by

$$\begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} := \begin{bmatrix} u & v \\ r & s \end{bmatrix} \begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix}.$$

Notice that the output satisfies

$$\begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} = -2^{60} M_{60}(\delta_{60(j-1)}, f_{60(j-1)}, g_{60(j-1)})$$

in the j -th Bigloop iteration.

6. lastuv: After the 10 bigloop iterations, the program completes computing the 600 divstep iterations. Therefore, the inputs of lastuv satisfy

$$\begin{bmatrix} \hat{u} & \hat{v} \\ \hat{r} & \hat{s} \end{bmatrix} = -2^{60} M_{60}(\delta_{540}, f_{540}, g_{540})$$

and

$$\tilde{f}[0] + 2^{60} \tilde{f}[1] = (f_{540} \bmod 2^{120}); \tilde{g}[0] + 2^{60} \tilde{g}[1] = (g_{540} \bmod 2^{120}).$$

By theorem, it is guaranteed that $f_{600} = 1$ or -1 . This subroutine computes f_{600} with $(\tilde{f}[0], \tilde{g}[0], \tilde{f}[1], \tilde{g}[1], \hat{u}, \hat{v})$ by

$$f_{600} := (\hat{u}(\tilde{f}[0] + 2^{60} \tilde{f}[1]) + \hat{v}(\tilde{g}[0] + 2^{60} \tilde{g}[1])) \gg 60.$$

Then, compute the output \check{u}, \check{v} by

$$(\check{u}, \check{v}) := (f_{600} \hat{u}, f_{600} \hat{v})$$

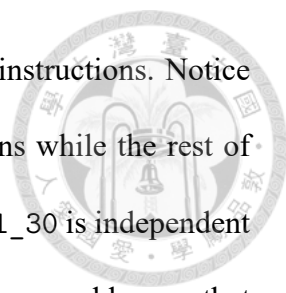
These are the subroutines that the computation is rather straightforward. Without using many CRYPTO LINE verification tricks, we verified these subroutines by specifying their precondition and postcondition as they are designed. Please see the verification code for more details.

5.6 Interleaving instructions

After verifying all the subroutines, there is still one more problem to deal with.

Notice that this implementation spends most of the execution time in the Bigloop that





loops 10 times. The loop body contains SISD instructions and SIMD instructions. Notice that the vectorized update, `vel_mul_30`, uses only SIMD instructions while the rest of the loop uses only SISD instructions, and the computation of `vel_mul_30` is independent of the rest of the loop. That being said, the set of registers and memory addresses that they affect are disjoint. Therefore, we can interleave the SISD and SIMD instructions for execution efficiency while maintaining equivalence.

A randomized algorithm is used to find an efficient permutation of the instructions while maintaining the execution sequence for the two independent parts. The speed record is achieved using this fast permutation. To verify this implementation, we model all the independent subroutines separately and verify them with CRYPTOLINE. Then, we just need to show that the permuted loop body is equivalent to the subroutines in order by inspecting the SSA form.

5.7 Results

We have verified all of the computations of the subroutines separately. Again, to show that the implementation will correctly compute modular inversion, we need to apply the theorem from the improved Bernstein-Yang Algorithm³ as mentioned in Section 2.3. To verify the theorems³ formally, which is also highly non-trivial, we leave it as future work as well.

³Such as $f_{600} = 1$ or -1 and the proof that implies 600 divstep iterations are enough.





Chapter 6 Concluding Remarks

6.1 Time Consumption

We use CRYPTO_{LINE} to verify the implementation in Chapter 4 and 5. All experiments are executed on an Ubuntu 20.04.2 LTS with 2.30GHz Intel Xeon and 755GB RAM with multi-thread enabled. Table B.1 and B.2 show the verification time. The simple subroutines can be verified in seconds while the complicated ones may take up to a couple of hours. This implies the verification engineer may wait for hours to get the verification result during debugging. Nonetheless, the time is still tolerable for verification.

Moreover, letting CRYPTO_{LINE} check the specified properties is not the most time-consuming task. To verify an assembly implementation, the verification engineer needs to first understand the algorithm and figure out the proper precondition/postcondition. Then, they should model the implementation delicately and apply the proper verification tricks. Despite the extensive workload, the writer managed to verify both implementations with the help of his instructor Bow-Yaw Wang, the programmer/inventor of the algorithm Bo-Yin Yang, and his research fellow Zih-Ming Li. It is hard to estimate the human time spent to verify the two implementations since it involves many people and the writer finished this work while studying for his Master's degree. What we can say is that it took around a year to verify the simple implementation in Chapter 4. After finishing verifying that, we

continued to verify the fast vectorized implementation in Chapter 5 in less than a month with the techniques learned from the previous experience.



6.2 The Verified Results

The strength of CRYPTO_{LINE} is verifying whether the assembly implementation matches the specification. In the two case studies in Chapter 4 and 5, we successfully verified that the assembly implementations behave the way they are designed on all inputs. Notice that the most common bugs on assembly code are unexpected overflowing because it is hard to detect this kind of bug by test cases. Our result ensures that this will not happen in this implementation. As for the correctness of the implementation as a whole, we still need the proof for the correctness of the algorithm which is mentioned in Section 4.4 and 5.7. To verify the proofs formally, we leave it as future work.

Combining the proof in [2] for the correctness and the verification of the implementation in Chapter 5, we can assure the correctness of the fast vectorized implementation. This makes our implementation the fastest verified implementation of modular inversion with $f_0 = 2^{255} - 19$ on Intel Skylake. With the verification techniques in this work, we believe it won't be hard to verify another implementation of Bernstein-Yang algorithm with different f_0 on different platforms.



References

- [1] D. J. Bernstein and B.-Y. Yang. Fast constant-time gcd computation and modular inversion. 2019(3):340–398, 2019. <https://tches.iacr.org/index.php/TCHES/article/view/8298>.
- [2] p. c. Daniel J. Bernstein and P. Wuille. safegcd-bounds. <https://github.com/sipa/safegcd-bounds>, 2021.
- [3] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 4-3-0 — A computer algebra system for polynomial computations. <http://www.singular.uni-kl.de>, 2022.
- [4] Y.-F. Fu, J. Liu, X. Shi, M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang. Signed cryptographic program verification with typed cryptoline. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19, page 1591–1606, New York, NY, USA, 2019. Association for Computing Machinery.
- [5] P. L. Montgomery. Modular multiplication without trial division. Mathematics of Computation, 44:519–521, 1985.
- [6] A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0. J. Satisf. Boolean Model. Comput., 9(1):53–58, 2014.





Appendix A — Proof

A.1 Proof about arithmetic precision

To explain why signed 64 bits registers is enough to compute divstep^{62} , we show the following theorem¹:

Theorem 1. $\mathcal{T}_n(\delta, f, g) = \mathcal{T}_n(\delta, f', g')$ if $f - f' \equiv g - g' \equiv 0 \pmod{2^n}$. Otherwise put, the 2-adic Bezout Coefficients depends on bottom bits.

Proof. We will use mathematical induction. It is clear that the condition holds for $n = 1$.

We take as a hypothesis that

$$\mathcal{T}_n(\delta, f, g) = \mathcal{T}_n(\delta, f \bmod 2^n, g \bmod 2^n) \text{ for } n \leq k - 1$$

We note that since δ_1 also depends only on $g \& 1$, we have

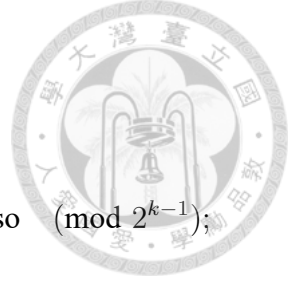
$$\mathcal{T}_k(\delta, f, g) = \mathcal{T}_{k-1}(\delta_1, f_1, g_1)$$

$$\mathcal{T}_k(\delta, f', g') = \mathcal{T}_{k-1}(\delta_1, f'_1, g'_1)$$

¹This proof is provided by Bo-Yin Yang.

We further notice that since $g \& 1 = g' \& 1$

$$f_1 - f'_1 = \begin{cases} f - f' & \text{if } \delta \leq 0 \text{ or } 2|g \\ g - g' & \text{if } \delta > 0 \text{ and } 2 \nmid g \end{cases} \equiv 0 \pmod{2^k}, \text{ hence also } \pmod{2^{k-1}};$$



and

$$2(g_1 - g'_1) = \begin{cases} (g - g') + (g \bmod 2)(f - f') & \text{if } \delta \leq 0 \text{ or } 2|g \\ (g - g') - (f - f') & \text{if } \delta > 0 \text{ and } 2 \nmid g \end{cases} \equiv 0 \pmod{2^k}.$$

or $g_1 - g'_1 \equiv 0 \pmod{2^{k-1}}$. Thus by the induction hypothesis, the desired result

holds. □



Appendix B — Table

B.1 Verification Time of the Simple Implementation

Table B.1 shows the time consumption by CRYPTOLINE to verify each subroutine in the simple x86 implementation rounded up in seconds. Multi-thread are enabled for speed. Safety check can be disabled when the algebraic properties are unused.

Table B.1: Verification time of the simple implementation.

| Subroutine Name | Safety(s) | Algebraic(s) | Range(s) | Total(s) |
|----------------------|-----------|--------------|----------|----------|
| fpadd25519 | 1 | 1 | 2 | 2 |
| fpcneg25519 | 1 | N/A | 1 | 1 |
| mul64xs64 | 5931 | N/A | 1 | 5932 |
| fpmul25519 | 1 | 1 | 37 | 38 |
| mul2x2s128_25519 | 285 | 2 | 341 | 627 |
| 1 divstep | N/A | N/A | 2 | 2 |
| Lemma 1 | N/A | 29109 | N/A | 29109 |
| jump64divsteps2_s255 | 3508 | 9 | 2068 | 5585 |

B.2 Verification time of the Fast Vectorized Implementation



Table B.2 shows the time consumption by CRYPTO LINE to verify each subroutine in the x86 fast vectorized implementation rounded up in seconds. Multi-thread are enabled for speed. Safety check can be disabled when the algebraic properties are unused.

Table B.2: Verification time of the fast vectorized implementation.

| Subroutine Name | Safety(s) | Algebraic(s) | Range(s) | Total(s) |
|------------------------|-----------|--------------|----------|----------|
| mod25519 | N/A | N/A | 4 | 4 |
| split9 | 1 | N/A | 2 | 2 |
| transition_portion | 1 | 1 | 1 | 1 |
| extract | 1 | 1 | 1 | 1 |
| updateuvrs | 1 | 1 | 10776 | 10776 |
| lastloop | 1 | 1 | 6253 | 6255 |
| lastuv | 1 | 1 | 1 | 1 |
| Each 1 divstep | N/A | N/A | 26 | 26 |
| A variation of Lemma 1 | N/A | 1 | N/A | 1 |
| vec_mul_30 | 1051 | 567 | 725 | 2343 |
| cneg | 30209 | 1 | 43 | 30252 |