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

密碼學函式庫實作中向量化基本操作的形式驗證

Formal Verification of Vectorized Implementations of Cryptographic Primitives

潘廣霖

Kuang-Lin Pan

指導教授: 陳偉松博士、王柏堯博士

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

中華民國 112 年 6 月

June, 2023

doi:10.6342/NTU202300977

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

密碼學函式庫實作中向量化基本操作的形式驗證

Formal Verification of Vectorized Implementations of Cryptographic Primitives

本論文係潘廣霖君（學號 R08922087）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 PAN KUANG LIN (student ID: R08922087) 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:

doi:10.6342/NTU202300977

Acknowledgements

Firstly, I would like to express my profound appreciation to my advisor, Dr. Bow-Yaw Wang, and my co-advisor, Dr. Tony Tan, for their continuous guidance and insightful feedback throughout the course of the research process. Their expertise and mentorship have been indispensable in shaping this thesis, and more importantly, in imparting the beauty of computer science to me.

Thanks should also go to instructors and partners I meet in FLOLAC (Formosan Summer School on Logic, Language, and Computation). My interest in functional programming and programming language design (PLD) would not have been sparked nor sustained without them.

Moreover, I am appreciative of my family for their support during these years, so I can pursue research without distractions. Lastly, I would like to recognize my dear friend Thôo-Tāu, for illumining my life during a challenging period.

Colophon

This thesis is proudly processed with Pandoc, a sophisticated document converter made by John MacFarlane.

doi:10.6342/NTU202300977

doi:10.6342/NTU202300977

摘要

現代密碼學函式庫中包含大量組合語言程式碼，無論是直接撰寫成，抑或是透過編譯器產生。其中，SIMD 指令常常做為提升效率的優化手段。如何有效率且簡約地使用手邊現有的工具來驗證這樣的程式碼，是正規驗證的一大課題。本論文拓展了模型檢查工具 CryptoLine，加入一系列的向量指令，並從實作後量子密碼學協議的函式庫中選取一些子程式來驗證其正確性和整數運算的安全性。驗證結果說明 Vector CryptoLine 採用的方法能順利建模來描述此類向量指令集在計算快速乘法時的行為，並且有潛力能被用來驗證未來其他後量子密碼學的函式庫。

關鍵字：形式驗證、密碼學、向量化運算

doi:10.6342/NTU202300977

doi:10.6342/NTU202300977

Abstract

Modern cryptographic libraries contain a large amount of assembly code derived either manually or through the use of compilers, where SIMD instructions are commonly used to reduce latency and increase speed. There is a challenge in verifying them efficiently and succinctly with the tools we have at our disposal. This thesis extends the model checking software, CryptoLine, with a set of vectorized instructions. Selected subroutines from various post-quantum cryptographic libraries are examined for correctness and integer safety. The results demonstrate that the approach adopted by Vector CryptoLine is successful in modelling these vectorized instructions in various forms of fast multi-limb multiplications, and has the potential to be used to validate other post-quantum cryptographic libraries in the future.

Keywords: formal verification, cryptography, vectorized computation

doi:10.6342/NTU202300977

doi:10.6342/NTU202300977

Contents

Page

口試委員會審定書 (Thesis Acceptance Certificate) **i**

Acknowledgements **iii**

摘要 (Abstract in Chinese) **v**

Abstract **vii**

Contents **ix**

List of Figures **xi**

List of Tables **xiii**

List of Listings **xv**

Chapter 1 Introduction **1**

Chapter 2 Background **3**

2.1 Multi-precision arithmetic 3

2.2 Fast Fourier transform 4

2.3 Lattice-based cryptography 5

2.4 Code-based cryptography 6

2.5 Design on domain-specific languages 6

Chapter 3 Overview of CryptoLine **9**

3.1 Syntax 10

3.2 Semantics 11

doi:10.6342/NTU202300977

3.3 Safety constraints 13

Chapter 4 Vector CryptoLine 15

4.1 Syntax extension 15

4.2 Design of Vector CryptoLine 16

4.3 Automatic variable aliasing 19

4.4 Case studies 20

4.4.1 Kyber NTT 20

4.4.2 SABER NTT 22

Chapter 5 Implementations 25

5.1 The additive FFT 25

5.2 The source code 27

5.3 Techniques used in McBits 29

5.4 Extraction of assembly code 31

5.5 Modeling the bit-shiftings 39

5.6 Generating the specifications 41

5.6.1 The post-condition 46

5.6.2 Implemention details 50

5.7 Verification result 51

Chapter 6 Conclusion 53

References 55

doi:10.6342/NTU202300977

List of Figures

Figure 3.1 The data flow of preparing and checking a CryptoLine model. . . 10

Figure 4.1 Type system of Vector CryptoLine 17
Figure 4.2 Typing rule schema of Vector CryptoLine 17
Figure 4.3 Typing rules for all combinations of vectorized move 18
Figure 4.4 An example of type annotation 18

doi:10.6342/NTU202300977

doi:10.6342/NTU202300977

List of Tables

Table 5.1 Running time of verifying the radix conversion subroutine 52

doi:10.6342/NTU202300977

doi:10.6342/NTU202300977

List of Listings

3.1 An example CryptoLine program solving Equation (3.1) 12

5.1 The radius conversion subroutine in McBits (saved as `rad.c`) 28

5.2 The driver code for `rad.c` 31

5.3 Commands to compile and invoke `itrace.py` 31

5.4 The assembly program of radix conversions 32

5.5 Sample of derived post-conditions in CryptoLine 48

doi:10.6342/NTU202300977

doi:10.6342/NTU202300977

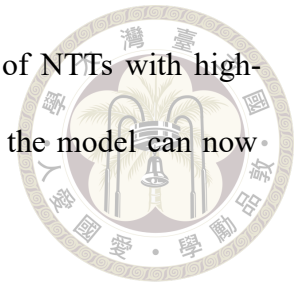
Chapter 1 Introduction

When implementing cryptographic libraries, programmers strive to ensure that the code is correct and efficient. This task gets more challenging in the context of post-quantum cryptography (PQC): to defend against the speedups of quantum computers, cryptographers have to cope with much larger amounts of data than traditional cryptosystems and with novel algorithms few have implemented or optimized, while protocols must stay efficient and secure.

Nowadays, it is common to see a significant amount of assembly code in sufficiently optimized software cryptographic libraries. Some implementations make extensive use of architecture-specific tricks in primitive constructs in order to maximize the capabilities of today's hardware. As the techniques evolve over time, it becomes increasingly difficult to reason on the correctness of highly-vectorized subroutines in reasonable amount of time and human intervention. By applying formal methods, developers can percisely define the intended properties of the software, and reason about its correctness rigorously.

In the efforts to formally verify these implementations, unrolling vector instructions by ad-hoc text substitutions is very labor-intensive and is proven tedious and error-prone. Having a reusable abstraction for vectors built into the framework itself is often handy to accurately model these esoteric instructions. Therefore, the formalization of vectors in CryptoLine arises. Several cryptographic primitives found in multiple PQC software

doi:10.6342/NTU202300977



are evaluated with this extended syntax. In particular, the proofs of NTTs with high-degree polynomials and large coefficients can be refactored so that the model can now be synthesized more concisely.

This thesis consists of three parts. In Chapter 2 and Chapter 3, the mechanism and design of CryptoLine are described, along with reviews on mathemetical foundation to cryptography for this thesis. Chapter 4 describes Vector CryptoLine's detailed design. Chapter 5 demonstrates the use of Vector CryptoLine to verify real cryptograhic primitives. In the end, a walkthrough on how to formally verify the additive FFT subroutine extracted from McBits is discussed.

doi:10.6342/NTU202300977

Chapter 2 Background

In the following sections, we briefly review some mathematical concepts. In the last section, we examine some aspects of DSL design.

2.1 Multi-precision arithmetic

The part of cryptographic libraries that is most critical to performance is primitive operations on various mathematical constructs, such as integers, group operations, points on elliptic curves. Much efforts and tricks are put into these cryptographic programs to maximize its throughput. A well-known operation yet notoriously hard to optimize is multiplication. Its asymptotic time complexity is kept being improved, but those intricate algorithms outperform only when the instance is astronomically large.

What is the scale of integers most cryptographic libraries deals with? For the public-key cryptography to guard against humans' current computing power, they typically lay around 2^{256} (approximately 10^{70}). For example, `Ed25519` operates on a prime field of $2^{255} - 19$ elements. Bitcoin's public-key cryptography operates on an elliptic curve `secp256k1` with $2^{256} - 2^{32} - 977$ elements. For PQC, the scale is even slightly larger. This draws a clear distinction from our everyday math, though, due to that most computers cannot hold operands in that scale in single registers.

doi:10.6342/NTU202300977

The textbook multiplication has a time complexity of $O(n^2)$ where n is the number of bits of input integers. For a long period of time, people believed that no better method would exist. The first improvement was Karatsuba algorithm (1962) with $O(n^{\log_2 3}) \approx O(n^{1.585})$. Further improvements, like the family of Toom-Cook algorithms (1963; 1969), were then developed [6,19]. In 1965, Cooley and Tukey discovered that discrete Fourier transform (DFT) could be done efficiently in a divide-and-conquer approach, known as fast Fourier transform (FFT) [10]. Pollard noticed (1971) that Cooley and Tukey's work directly yielded a faster algorithm for integer multiplications, what the so-called number-theoretic transform (NTT) [15].

People found out that despite the overhead of NTT was larger then Toom-Cook's, NTT implementations, along with various optimization tricks applied, could be much efficient on modern computer hardware. It is a trend that cryptographic algorithms decide to include NTT in the specifications and reference implementations.

2.2 Fast Fourier transform

A fast Fourier transform (FFT), in cryptographic (rather than digital signal processing) context, is an efficient algorithm that is able to transform a polynomial from its coefficients to evaluations on some chosen points, and vise versa. The primary purpose of FFT is to calculate the product of two large numbers or polynomials, since the result of polynomial multiplications in this form is simply the point-wise products.

The core idea of the widely-used base-2 Cooley-Tukey FFT is that an n-coefficient polynomial $f(x)$ can be written into two half-sized polynomials by grouping coefficients of odd and even degrees, respectively. This operation yields two smaller polynomials

doi:10.6342/NTU202300977

about x^2, namely,

$$f(x) = f^0(x^2) + xf^1(x^2).$$

This operation can be applied recursively on f^0 and f^1 for $\lceil \log_2 n \rceil$ times with paddings. The eventual result is a sequence of constants evaluated at roots of unity in complex domain. It can be shown that an inverted process can combine those evaluations back together, recovering the original coefficients.

The efficiency of Cooley-Tukey algorithm comes from the so-called butterfly structure between the chosen roots. Suppose f is an $(n+1)$-coefficient polynomial. For a given value c, recursing into $f^0(c^2)$ and $f^1(c^2)$ enables one to compute both $f(c)$ and $f(-c)$ with only one additional addition or substraction for each:

$$\begin{aligned} f(c) &= f^0(c^2) + c\, f^1(c^2); \\ f(-c) &= f^0(c^2) - c\, f^1(c^2). \end{aligned} \tag{2.1}$$

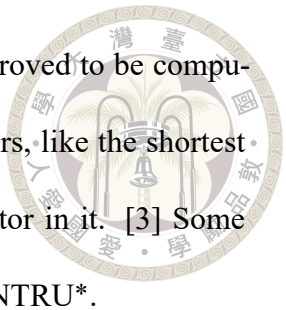
Each step cuts the number of terms by half, with $O(n)$ multiplications and additions in each layer. The overall number of arithmetic operations is bound by $T(n) \leq 2\,T(n/2) + O(n)$, or $T(n) = O(n \log n)$, making it possible to be compute in quasilinear time. This fact draws public attention to the development of other variants of FFT [10].

2.3 Lattice-based cryptography

Lattice-base cryptography is the cryptography based on the concept of lattices. A lattice is the sequence of linearly-independent vectors $v_1, v_2, \cdots, v_n$ spanned with integer coefficients:

$$\{a_1 v_1 + a_2 v_2 + \cdots + a_n v_n : (a_1, \cdots, a_n) \in \mathbb{Z}^n\}.$$

doi:10.6342/NTU202300977



There are several fundemental problems about lattices that are proved to be computationally intractable to solve, even with the aid of quantum computers, like the shortest vector problem (SVP): given a lattice, find the shortest nonzero vector in it. [3] Some lattice-based cryptographic systems are Kyber [16], SABER [5] and NTRU*.

2.4 Code-based cryptography

Code-based cryptography is a kind of cryptosystem whose security depends on the hardness of decoding an error-correcting code (ECC). The concept can be roughly depicted as follows: A public key admits an error-correcting code to encode messages, but without knowing the private key it is believed to be computationally intractable to eliminate the noises (the *syndromes*) in the payload.

McEliece [14] is the pioneered public-key cryptosystem (PKC) based on coding theory. The choice of binary Goppa code make it resistant to efficient structural attacks. There exist PKCs that use code systems other than Goppa code. For instance, Niederreiter PKC uses Generalized Reed-Solomon (GRS) code. Classic McEliece is a post-quantum cryptosystem based on McEliece and Niederreiter [18].

2.5 Design on domain-specific languages

The need of assembly-level static analysis tools arises from digital forensic area, where researchers study proprietary program samples, often malwares, without actually executing them. Among the surveyed tools, there are also analysis tools that can understand SIMD instructions accurately. angr [2], an open source symbolic execution framework,

* the specification can be accessed at https://ntru.org/f/ntru-20190330.pdf.

doi:10.6342/NTU202300977

supports unpacking AVX2 instructions since version 8. K semantic framework, an executable semantic framework, is capable of modelling semantics of assembly code (specifically, x86_64 [8]). In formal verification area, the importance of the ability to model SIMD instructions for high-performance cryptographic library is also addressed in section 4 of [21]. I took inspirations on the architectures from those frameworks to craft Vector CryptoLine.

CryptoLine is authored in OCaml, an object-oriented dialect of the ML programming language family. Being a functional language, it features a well-established type inferring algorithm, and compiles to efficient native code. One advantage of embracing a functional programming paradigm is to define a strongly-typed abstract syntax tree (AST), and then benefit from the sophisticate pattern-matching syntax in OCaml on tree node types. Unfortunately, this makes the AST so strict that even a tiny change propagates to the entire code base. Extending in later passes, on the other hand, requires duplication of type definitions.

To find a balance between type safety and flexibility of experimenting, I surveyed some meta-programming techniques, and find nanopass [17] particularly promising. Nanopass is a multi-pass compiler framework that incrementally refines the AST between passes. This turns out to be a generic technique for DSLs to produce intermediate representations (IR). For example, one can define a high-level statement, and write a pass to transform it to equivalent multiple low-level statements, and the type safety is enforced by the framework throughout the course of code generation.

doi:10.6342/NTU202300977

doi:10.6342/NTU202300977

Chapter 3 Overview of CryptoLine

CryptoLine is a model checker that is specialized on assembly-level cryptographic libraries. It combines the power of both computer algebra systems (CAS) and satisfiability modulo theories (SMT) solvers to verify whether assembly programs satisfy the specified mathematical properties. CryptoLine users (the *verifiers*) first export or extract the assembly source and translate it to the eponymous Cryptoline DSL. After the preparation of the source code, verifiers are supposed to interact with the model checkers repeatedly: If the model checker gets stuck, the verifiers have to annotate the DSL to provide more information and/or examine the output produced by the checkers. The correctness of the program is rigorously verified if it passes all the checks, or a counterexample is displayed for further diagnosis. The data flow is outlined in Figure 3.1.

To aid the translation of assembly code into Cryptoline DSL, some tool scripts are distributed along with CryptoLine. The script `itrace.py` exports the assembly trace and values in registers from a running program via a debugger (here GDB); The script `to_zdsl.py` is capable of doing text replacements following some in-file translation rules to transpile the extracted assembly instructions to Cryptoline DSL. As a result, users of CryptoLine can benefit greatly from those tools to minimize the one-off annotations they have to write or generate, and focus more on the specifications they are deriving.

doi:10.6342/NTU202300977

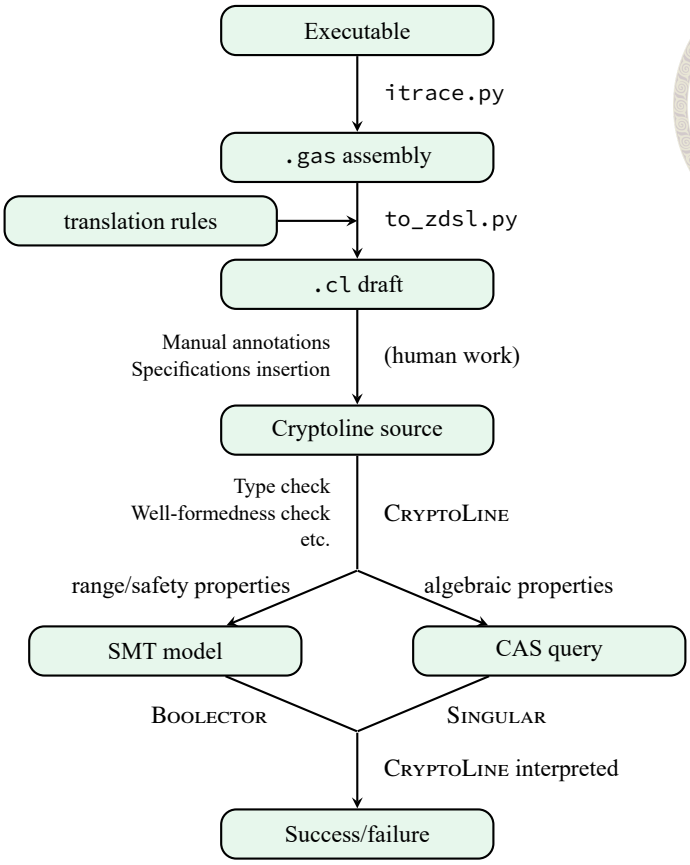


Figure 3.1: The data flow of preparing and checking a CryptoLine model.

3.1 Syntax

Cryptoline is statically and strongly typed. All integral constants must be annotated with their bit widths and signs. For example, `uint8` denotes the type of unsigned 8-bit integers. Cryptoline instructions resemble assembly instructions in common instruction set architectures (ISAs) for assembly programs to be converted into in 1-to-1 in most cases. A complete Cryptoline model consists of the function signature, the pre-condition, the program body, and the post-condition. Besides a pre-condition and post-condition, it is possible to specify predicates in the middle of a program body with instructions like `assume`, `assert`, `ghost` or `cut`. The usage and semantics of these instructions are explained in later chapters.

doi:10.6342/NTU202300977

3.2 Semantics

Each instruction in CryptoLine is interpreted separately in terms of polynomials (algebraic property) and fixed-size bit vectors (range property). The two different mathematical structures, corresponding to two kinds of tactics, can be used alternatively to prove the desired properties, which are written as specifications. The former ones are processed by a computer algebra system (CAS), and the latter ones are processed with an SMT solver.

An SMT solver proves a specification by asserting that the negation of it is unsatisfiable. If it is not the case, it means that there exists an input satisfying the pre-condition that violates the post-condition. A capable SMT solver can further output the certificate or counterexample in either cases.

A CAS solver, on the other hand, solves an instance of polynomial entailment problem. Each algebraic operation is encoded as an input polynomial, and one asks if the specification, also encoded as a polynomial, can be *entailed* from all input polynomials. In an algebraic context, a commutative ring is constructed from the input ideals, and the CAS shows us whether a specified polynomial is an element of this ring or not. That is, equivalently, whether the specification can be yielded from input ideals with additions and multiplications. The following example describes how an algebra problem can be converted into an instance of ideal membership problem, and being proved with CryptoLine.

doi:10.6342/NTU202300977

Example 3.1 (ideal membership problem)**.** Given $x, y, z \in \mathbb{Z}$ and constraints

$$\begin{cases} x + y + z &= 1 \\ x^2 + y^2 + z^2 &= 2 \\ x^3 + y^3 + z^3 &= 3 \end{cases}$$

proving that $x^5 + y^5 + z^5 = 6$ is equivalent to checking the following:

$$\begin{gathered} x^5 + y^5 + z^5 - 6 \overset{?}{\in} \\ \mathbb{Z}[x, y, z]/\langle x + y + z - 1, x^2 + y^2 + z^2 - 2, x^3 + y^3 + z^3 - 3 \rangle. \end{gathered} \tag{3.1}$$

The statement is true. We can prove the fact with the following void CryptoLine program:

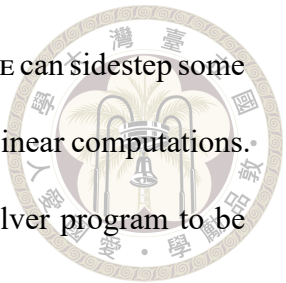
Listing 3.1: An example CryptoLine program solving Equation (3.1)

```
1  proc main(uint8 x, uint8 y, uint8 z) =                    input variables
2  { and [                                                   pre-conditions
3      x + y + z = 1,
4      x**2 + y**2 + z**2 = 2,
5      x**3 + y**3 + z**3 = 3
6    ] && true }
7    nop;                                                    program body
8  { x**5 + y**5 + z**5 = 6 && true }                        post-conditions
```

Under most circumstances, variables are implicitly connected with instructions. CryptoLine is responsible for collecting these ideals automatically. However, there are some instructions like bitwise shifts that cannot be modelled with polynomials. By `assume`-ing an outcome, an ideal is constructed and put into quotient ring explicitly, though one have to make sure the proposition indeed holds to avoid inconsistencies.

Compared to traditional verification frameworks that solely rely on an SMT solver as its theorem prover, adopting to an algebra system makes the verification experience on CryptoLine more flexible and time-efficient. While SMT instances are more generic, by

doi:10.6342/NTU202300977



modeling modular equations into polynomials, verifiers of CryptoLine can sidestep some of pathological instances for SMT solvers, like large instances of non-linear computations. CryptoLine is also capable of sending out queries to a different solver program to be neutral on the verifier we select, though the default ones — Singular for CAS and Boolector for SMT solver*, respectively — typically give the best performance.

3.3 Safety constraints

For the soundness that all algebraic properties rely upon, CryptoLine needs to check all safety conditions involving in each applicable instruction beforehand. For example, an addition must either store a carry flag, or guarantee that it never overflows. In the latter case, it is expected that the "possiblely overflow" property should be proven UNSAT. Another example is that a casting to a narrower type does not change the value it represents (a *value-preserving* cast). The algebraic semantics is sound to CryptoLine if and only if the safety condition is not violated.

Given a program, CryptoLine synthesizes and checks the safety property automatically. Safety conditions are aggregated from all instructions, and then the conjunction of them is sent to an SMT solver. The verifier should notice that a subroutine may generate thousands of safety conditions in practice, and this one-shot approach highly likely gets stuck. CryptoLine offers a `-isafety` option to verify them in an incremental fashion, with configurable timeout and parallelism to check the model efficiently. This is the more effective approach in most realistic cases.

* External dependencies Singular can be accessed at https://www.singular.uni-kl.de/, and Boolector can be accessed at https://boolector.github.io/.

doi:10.6342/NTU202300977

doi:10.6342/NTU202300977

Chapter 4 Vector CryptoLine

In this chapter, I present Vector CryptoLine (abbreviated VCL), an extension to CryptoLine that provides single-instruction-multiple-data (SIMD) semantics. It offers typing support for fixed-size homogeneous arrays ("vectors") that can concisely express SIMD data models in popular CPU instruction sets, e.g., x86_64 SSE, AVX, AVX2, or ARM Neon. Moreover, it provides a concise way to permute elements within a vector, in the same way bits are shuffled within registers, in which the original instructions of CryptoLine cannot express sequentially without usage of temporary variables.

4.1 Syntax extension

By design, Vector CryptoLine (VCL) translates one SIMD instruction into one or a few statements, resulting shorter code than ones in original CryptoLine, and this may imply tighter feedback loops when debugging. Under the hood, Vector CryptoLine is an extension on top of the original syntax. During the survey, the path nanopass had taken seemed to be a perfect solution to this. Unfortunately, nanopass was originally designed for Scheme and was ported to Racket, and to implement a similar metaprogramming framework in OCaml would largely depend on the internal workings of the compiler. To keep things safe, the change to data structure should be minimal. This is why VCL starts

doi:10.6342/NTU202300977

as a refactor on the originally-closed type definitions for the vector part to be implemented as an additional pass. It is a trade-off result between macro expansions and maintaining its own version of AST alongside the scalar one.

Vector CryptoLine was designed with reusability in mind: A set of translation rules written for one procedure could be applied to other procedures using the same set of SIMD instructions. We experimented on selected subroutines verified by original CryptoLine to test its expressiveness and performance. Compared with prior works modeled with only scalars, the resulting code from VCL is terser, equally type-safe, and can be verified in near time and memory usage.

4.2 Design of Vector CryptoLine

In Cryptoline DSL, every integral value has a fixed bit size, and is either a signed integer `sint` or an unsigned integer `uint`. In addition, `bit` is a shortand of `uint1`, as 1-bit integers are typically modelled after binary flags. These types, called scalar types, are sufficient to model integral arithmetic and various boolean flags in typical CPUs. Vector CryptoLine expands the type system by introducing types for vectors, namely vector types. A vector type is a product type of some scalar type and a positive integer indicating its dimension. For example, `uint8[16]` is a vector type over scalars `uint8` containing 16 elements, and `bit[32]` describes a bit-vector of 32 bits wide. The extended type system is shown in Figure 4.1.

Vectors are manipulated with an exclusive set of instructions. For convenience, the names of their counterparts in scalar CryptoLine are reused. This concept is commonly

doi:10.6342/NTU202300977

$$\frac{\tau : \mathsf{Type}}{\tau : \mathsf{SclType}}\ (\text{T-Scl}) \qquad \frac{\tau[\cdot] : \mathsf{Type}}{\tau[\cdot] : \mathsf{VecType}}\ (\text{T-Vec})$$

$$\frac{b : \mathbb{N}}{\mathsf{uint}\ b : \mathsf{SclType}}\ (\textsc{Uint}) \qquad \frac{n : \mathbb{N} \setminus \{1\}}{\mathsf{sint}\ b : \mathsf{SclType}}\ (\textsc{Sint}) \qquad \frac{\tau : \mathsf{SclType} \quad n : \mathbb{N}}{\tau[n] : \mathsf{VecType}}\ (\textsc{Vec})$$

Figure 4.1: Type system of Vector CryptoLine

$$\frac{n : \mathbb{N} \qquad \Gamma \vdash \mathsf{\%a} : \tau[t]}{\Gamma, \mathsf{\%v} : \tau[n] \vdash \mathsf{broadcast\ \%v\ n\ \%a}}\ (\textsc{Broadcast}) \qquad \frac{\Gamma \vdash a : \tau[n]}{\Gamma, v : \tau[n] \vdash \mathsf{mov}\ v\ a}\ (\text{VMov})$$

$$\frac{\Gamma \vdash a : \tau[n] \qquad \Gamma \vdash b : \tau[n] \qquad \mathsf{instr} \in \{\mathsf{add}, \mathsf{sub}, \mathsf{mul}\}}{\Gamma, v : \tau[n] \vdash \mathsf{instr}\ v\ a\ b}\ (\text{VUniBin})$$

$$\frac{\Gamma \vdash a : \tau[n] \qquad \Gamma \vdash b : \tau[n]}{\Gamma, s : \mathsf{bit}[n], v : \tau[n] \vdash \mathsf{adds}\ s\ v\ a\ b}\ (\text{VAdds})$$

$$\frac{(T\ b), (T\ (b-c))\text{: SclType} \quad \Gamma \vdash a : (T\ b)[n] \qquad c : \mathbb{N} \qquad \Gamma \vdash c < b}{\Gamma, v_h : (T\ (b-c))[n], v_l : (T\ c)[n] \vdash \mathsf{split}\ v_h\ v_l\ a\ c}\ (\text{VSplit})$$

$$\frac{\Gamma \vdash a : \tau[n]}{\Gamma, v : \sigma[n] \vdash \mathsf{cast}\ []\ v@\sigma[n]\ a}\ (\text{VCast})$$

$$\frac{(\text{T b})\text{: SclType} \quad \Gamma \vdash a : (T\ b)[n] \qquad \Gamma \vdash b : (T\ b)[n]}{\Gamma, v_h : (T\ b)[n], v_l : (\mathsf{uint}\ b)[n] \vdash \mathsf{mull}\ v_h\ v_l\ a\ b}\ (\text{VMull})$$

Figure 4.2: Typing rule schema of Vector CryptoLine

known as *method overloading* in most OOP languages, such as Java or C#. In the scope of our verification targets, we present the typing rules for VCL in Figure 4.2.

VCL provides two kinds of syntax in order to construct vectors. One is to enclose a comma-separated list of scalars in a pair of square brackets to form a vector *literal*, and the other is to use instructions that produce a vector, and specify a vector *variable* as the destination. They can be used interchangably in instructions. An example of refined typing rules is given in Figure 4.3. Vector variables are stored separately from scalar variables by prefixing a "%" symbol to their names to ensure that their names will never clash.

doi:10.6342/NTU202300977

$$\frac{\Gamma \vdash \%\mathsf{a} : \tau[n]}{\Gamma, \%\mathsf{v} : \tau[n] \vdash \mathsf{mov}\ \%\mathsf{v}\ \%\mathsf{a}}\ \text{(VMov-VV)}$$

$$\frac{\Gamma \vdash \mathsf{a}_1 : \tau \qquad \cdots \qquad \Gamma \vdash \mathsf{a}_n : \tau}{\Gamma, \%\mathsf{v} : \tau[n] \vdash \mathsf{mov}\ \%\mathsf{v}\ [\mathsf{a}_1, \mathsf{a}_2, \cdots, \mathsf{a}_n]}\ \text{(VMov-VL)}$$

$$\frac{\Gamma \vdash \%\mathsf{a} : \tau[n]}{\Gamma, \mathsf{v}_1 : \tau, \cdots, \mathsf{v}_n : \tau \vdash \mathsf{mov}\ [\mathsf{v}_1, \mathsf{v}_2, \cdots, \mathsf{v}_n]\ \%\mathsf{a}}\ \text{(VMov-LV)}$$

$$\frac{\Gamma \vdash \mathsf{a}_1 : \tau \qquad \cdots \qquad \Gamma \vdash \mathsf{a}_n : \tau}{\Gamma, \mathsf{v}_1 : \tau, \cdots, \mathsf{v}_n : \tau \vdash \mathsf{mov}\ [\mathsf{v}_1, \cdots, \mathsf{v}_n]\ [\mathsf{a}_1, \cdots, \mathsf{a}_n]}\ \text{(VMov-LL)}$$

Figure 4.3: Typing rules for all combinations of vectorized move

$$\frac{\Gamma \vdash \mathsf{a} : \tau \qquad \Gamma' \vdash \mathsf{mov\ v\ a}}{\Gamma' \vdash \mathsf{mov\ v@}\tau\ \mathsf{a}}\ \text{(Mov-Hinted)}$$

$$\frac{\Gamma \vdash \%\mathsf{a} : \tau[n] \qquad \Gamma' \vdash \mathsf{mov}\ [\mathsf{v}_1, \mathsf{v}_2, \cdots, \mathsf{v}_n]\ \%\mathsf{a}}{\Gamma' \vdash \mathsf{mov}\ [\mathsf{v}_1, \mathsf{v}_2, \cdots, \mathsf{v}_n]@\tau[\mathsf{n}]\ \%\mathsf{a}}\ \text{(VMov-LV-Hinted)}$$

Figure 4.4: An example of type annotation

When a VCL model is being checked, each vector literal is type-checked to ensure that the vector contains exact number of elements, and scalars in a vector are always of the same type. Instructions are type-checked to ensure that the type annotations, if given, matches the input or output type of instructions, as shown in Figure 4.4. VCL in CryptoLine works like a single pass to a multi-pass compiler: If the type checks, VCL erases redundant type annotations on individual elements and transforms them into scalar statements. After the type checking, the compiler generates scalars without type hintings, to avoid tapping in unnecessary type assertions again. The verification then proceeds as if the abstract syntax tree (AST) were only consist of scalars.

doi:10.6342/NTU202300977

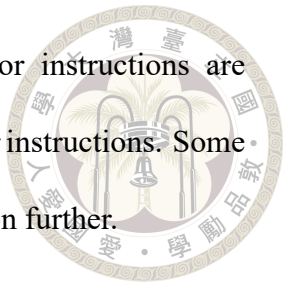
4.3 Automatic variable aliasing

A large part of vector instructions are invented for shuffling around two registers. Few of them has concise description. For example, a `ymm` register in x86_64 AVX2 is 256-bit wide, and instructions `vpunpckhqdq`/`vpunpcklqdq` take two `ymm`s `v0` and `v1` and split both registers into 64-bit lanes. The lanes are then interleaved and placed to the destination register, also split to the same width of lanes. Such instructions are used extensively in FFT subroutines to reorder elements in-place after each iteration. As we can see, the plain-text description is already very complicated, let alone the formal specifications.

A substantial feature in VCL is its automatic variable aliasing designed for modeling various permutations in vector registers concisely. This can be a problem when the variables reading and writing are overlapped. The simplest example in beginner lesson of programming would be an instruction swapping two elements in a vector, namely, `mov [b, a] [a, b];`. In a naive implementation, it translates to `mov a b; mov b a;`, but the semantic would be incorrect. VCL handles this scenerio transparently by detecting and inserting additional temporary variables to preserve the correct semantics.

To achieve this, we represent each operations as a pair (w, rs), where rs is a set of variables that are being read from, and w is some variable that is be written to. The notion is that reads can happen simultaneously as they do not modify the state, while writes must be sequenced. We maintain a set of "clobbered" variables T, initially empty, and scan through the sequence of operations. If we find any variable $v \in rs$ is also in T, we know we need an alias for v. After each operation, merge rs into T. For the most basic swapping example, the sequence is $(b, [a]), (a, [b])$, and we can see that a is clobbered in the first operation while being read in the second operation, so it needs an alias.

doi:10.6342/NTU202300977



This feature abstracts away from verifiers the fact that vector instructions are inevitably sequential in VCL, reducing the friction of translating vector instructions. Some heuristics may be applied to cut down the number of aliasing operation further.

4.4 Case studies

Two proposed PQC software have x86_64 AVX2 implementations. In this section, their NTT are translated to Vector CryptoLine for assessment.

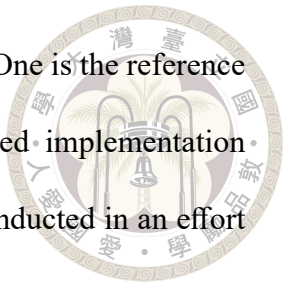
4.4.1 Kyber NTT

CRYSTALS* is a lattice-based PQC system. Its KEM algorithm, CRYSTALS-KYBER (Kyber [16]), was one of the finalists in the third round of the *Post-Quantum Cryptography Standardization* project held by NIST, under the *Public-key Encryption and Key-establishment Algorithms* category.

Kyber's security is based upon the hardness of Ring-LWE (ring learning with errors) problems. In its round 2 specification and onward, it computes multi-limb multiplications using negacyclic NTTs under the polynomial ring $\mathbb{F}_q[x]/\langle x^{256}+1\rangle$ where $q = 3329 = 1 + 13 \times 256$. The NTT consists of 7 layers, each layer split the polynomial into two equally-sized ones, eventually factorize a ring element into remainders of $(x^2-\zeta), (x^2-\zeta^3), \cdots, (x^2-\zeta^{255})$ with generator $\zeta = 17$. Base elements in $\mathbb{F}_q$ are center-shifted to $\pm(q-1)/2$, and Montgomery reductions are applied to each layer to keep all field elements representable in the 16-bit signed type throughout the calculation.

* The software package can be accessed at https://pq-crystals.org/.

doi:10.6342/NTU202300977



Kyber includes two implementations in its submission to NIST. One is the reference implementation in x86_64 assembly, and the other is an optimized implementation utilizing the AVX2 instruction set. Most experiments on VCL are conducted in an effort to the NTT sub-routine in the optimized version.

Despite Kyber's structure of NTT being arguably the simplest for a PQC specification, we mention several implementation details that pose challenges to CryptoLine:

- Kyber's preference on signed integer arithmetic is uncommonly seen in other cryptosystem libraries. Unsigned integers are much easier to reason about at programming. For example, in the C programming language, the overflowing behavior of unsigned integers is specified, but the signed counterpart is left undefined; i.e. C code containing signed integer overflows is not considered valid.
- Kyber uses Montgomery reductions, and the use of lazy reductions make it difficult to give a tight bound for range assertion. A loose bound at earlier stages means a looser bound in a subsequent stage, and when the ranges are too loose, it is harder to prove that the number are bounded.
- Seven layers of non-linear computations is very time and memory consuming to be verified end-to-end, if not infeasible, requiring us to cut the program into several pieces and write out detailed conditions in between, which means the verifiers need to know exactly the memory layout at each stage.

With both adding new features to CryptoLine and improving the proof techniques, we managed to prove the correctness of Kyber's optimized NTT procedure. However, we would like to emphasize on the reasoning on vectorization within each layers. The range

doi:10.6342/NTU202300977

post-conditions were defined elsewhere, and the full algebraic specifications were later proposed and verified with the non-local reasoning technique in [11].

4.4.2 SABER NTT

SABER* (or Saber) is another lattice-based PQC system. It was one of the candidates in the third round of the *Post-Quantum Cryptography Standardization* project held by NIST, under the *Public-key Encryption and Key-establishment Algorithms* category.

Saber's security is based upon the hardness of Mod-LWR (module learning with rounding) problem. Saber operates on the polynomial ring $\mathbb{F}_q[x]/(x^{256}+1)$ with $q = 2^{13}$ a power of 2. Saber orginially performed polynomial multiplications via Toom-Cook multiplications. An NTT approach was mentioned in of the original paper of Saber (pp. 16 of [7]) date back to 2018, and was adopted as the standard method later.

Because Saber's choice of q itself is not a prime, it is not NTT-friendly [5]. Saber uses a trick to compute multiplications: It finds two primes $q_0 = 7681$ and $q_1 = 10753$ such that $q_0 q_1 > q^2$, maps each element of its base field x to a pair $(x \bmod q_0, x \bmod q_1)$ and perform NTT multiplications on those fields respectively. It then recovers the result $x \bmod (q_0 q_1)$ by Chinese Remainder Theorem (CRT). This always does the trick since each of the resulting coefficient is less than q^2, and the exact values are kept when taking modulo of $q_0 q_1$.

Two assembly subroutines `sabermul_nttmul_poly_ntt` and `speed256nx3_nttmul_poly_basemul_montgomery` for fields of order q_0 and q_1, respectively, are modelled from Saber's optimized AVX2 implementation to assess

* The software package can be accessed at https://www.esat.kuleuven.be/cosic/pqcrypto/saber/resources.html.

doi:10.6342/NTU202300977

on the expressiveness of VCL. These subroutines were already passing CryptoLine's verification [11]. With these scalar models in reference, the translation recipe was written to transpile Saber's SIMD instructions and permutations. The resulting VCL sources also passed the verification.

doi:10.6342/NTU202300977

doi:10.6342/NTU202300977

Chapter 5 Implementations

This chapter illustrates the process of verifying a real-world SIMD subroutine during and after the making of Vector CryptoLine with an additive FFT subroutine in PQC software. An additive FFT is used in an implementation of Classic McMeliece to decode syndromes (see Section 2.4).

5.1 The additive FFT

From an algebraic point of view, the Cooley-Tukey's FFT algorithm can also be seen as the factorization of $x^{2^n} - 1$ over complex domain. NTT generalizes the idea to any finite fields containing an 2^n-th root, without involving complex numbers. Powers of that 2^n-th root form a multiplicative group of order 2^n, a fast Fourier transform is therefore said to be *multiplicative* [9]. In Gao and Mateer's paper on additive FFT, they further specialize the idea exploiting the properties of fields of characteristics 2.

Every finite field with characteristic 2 corresponds to a specific extension field GF_{2^n}. Two of the fundamental properties of such field are that $a = -a$ for every field element a, and $(x + y)^2 = x^2 + y^2$ for every pair of field elements x and y. This kind of fields is of particular interest to computer scientists, since they are well-suited for modern computers and circuits where field elements can be stored and manipulated efficiently as bit-vectors

doi:10.6342/NTU202300977

(e.g., addition is always equivalent to bitwise-XORs). On the other hand, the lack of any 2^n-th root makes traditional radix-2 FFT unavailable. FFT over other radixes can still be performed, but the increased complexity and overhead often makes it unworthwhile [1].

Similar to FFT's decomposing with x^2, Gao and Mateer choose to expand a polynomial into $(x^2 + x)$ instead, i.e., to write a $2n$-coefficient polynomial $f(x)$ in the form

$$(c_0 + c_1 x)(x^2 + x) + (c_2 + c_3 x)(x^2 + x)^2 + \cdots + (c_{2n-2} + c_{2n-1})(x^2 + x)^{n-1}, \quad (5.1)$$

Gao and Mateer call it a *Taylor expansion*. By the fact that $(\alpha + 1)^2 + (\alpha + 1) = \alpha^2 + \alpha$ under any field of characteristic 2, it follows that

$$\begin{aligned} f(\alpha) &= f^0(\alpha^2 + \alpha) + \alpha f^1(\alpha^2 + \alpha); \\ f(\alpha + 1) &= f^0(\alpha^2 + \alpha) + (\alpha + 1) f^1(\alpha^2 + \alpha) \\ &= f(\alpha) + f^1(\alpha^2 + \alpha). \end{aligned} \quad (5.2)$$

Observe that overlapped computations, in this case, a multiplication between $\alpha + 1$ and $f^1(\alpha^2 + \alpha)$, can be saved to yield both $f(\alpha)$ and $f(\alpha + 1)$, exhabiting a butterfly scheme similar to a multiplicative FFT. At the bottom level of recursion, it remains a multiplication and an addition evaluating linear terms.

Compared from traditional FFT's multiplication, the result of any layer from additive FFT can be seen as evaluations over all subset sums of basis at that layer. It begins with the monomial basis in $GF_{2^n}[z]/f(z)$ for some irreducible polynomial $f(z)$ of degree n, say $\{z^{n-1}, z^{n-2}, \cdots, z, 1\}$, and each layer of radix conversion maps k to $k^2 + k$ (1 maps to 0 and thus dropped). To be able to perform this operation repeatedly, a variable substitution should be carried out to the sequence of basis so that the last term becomes 1. This procedure is known as *twisting*.

doi:10.6342/NTU202300977

A closer look to the basis throughout the subroutine results in a pattern as follows:

$$
\begin{aligned}
&\{\ z^{n-1} &&, \cdots, z^4 && , z^3 && , z^2 && , z && , 1\ \} \\
\xrightarrow{k\mapsto k^2+k}&\{\ z^{2n-2}+z^{n-1} && , \cdots, z^8+z^4 && , z^6+z^3 && , z^4+z^2 && , \beta_0 (= z^2+z) && \} \\
\xrightarrow{\cdot\beta_0^{-1}}&\{\ \frac{z^{2n-2}+z^{n-1}}{z^2+z} && , \cdots, \frac{z^8+z^4}{z^2+z} && , \frac{z^6+z^3}{z^2+z} && , \frac{z^4+z^2}{z^2+z} && , 1 && \} \\
\xrightarrow{k\mapsto k^2+k}&\{\ \cdot && , \cdots, \cdot && , \cdot && , \beta_1 && && \} \\
\xrightarrow{\cdot\beta_1^{-1}}&\{\ \cdot/\beta_1 && , \cdots, \cdot/\beta_1 && , \cdot/\beta_1 && , 1 && && \} \\
&\ \vdots
\end{aligned}
\tag{5.3}
$$

where each $k \mapsto k^2 + k$ is a radix conversion, and multiplications with scaling factors $\beta_0^{-1}, \beta_1^{-1}, \cdots, \beta_{n-1}^{-1}$ are twistings. The process can stop early if the polynomial already has degree one.

5.2 The source code

In McBits [1,4], an optimized implementation to the Classic McEliece cryptosystem, the additive FFT was used to decode the syndrome in the underlying code system. The formal verification of radix conversion procedure in McBits is presented here to demonstrate the ability that CryptoLine can not only understand a limited set of instructions written by human, but can also, when utilized by the verifier, handle moderately optimized assembly code generated from a modern compiler.

The subroutine we intend to verify is extracted from the software package McBits included in SUPERCOP [20]. We attempted various techniques translating the instructions into algebraic and range queries, and make use of VCL to deal with bitwise shiftings. At the end of this chapter, we verify this implementation to be correct.

doi:10.6342/NTU202300977

Listing 5.1: The radius conversion subroutine in McBits (saved as `rad.c`)

```c
#include <inttypes.h>
#define GFBITS 12

extern void PQCLEAN_MCELIECE348864_AVX_vec_mul_asm(uint64_t *, const uint64_t
  *, const uint64_t *);

void PQCLEAN_MCELIECE348864_AVX_vec_mul(uint64_t *h, const uint64_t *f, const
  uint64_t *g) {
  PQCLEAN_MCELIECE348864_AVX_vec_mul_asm(h, f, g);
}
void radix_conversions(uint64_t *in) {
    int i, j, k;
    const uint64_t mask[5][2] = {
      /* ... constants ... */
    };

    const uint64_t s[5][GFBITS] = { /* ... constants ... */ };

    for (j = 0; j <= 4; j++) {
        for (i = 0; i < GFBITS; i++) {
            for (k = 4; k >= j; k--) {
                in[i] ^= (in[i] & mask[k][0]) >> (1 << k);
                in[i] ^= (in[i] & mask[k][1]) >> (1 << k);
            }
        }
        PQCLEAN_MCELIECE348864_AVX_vec_mul(in, in, s[j]); // scaling
    }
}
```


doi:10.6342/NTU202300977

5.3 Techniques used in McBits

The radix conversion task is performed on a quotient ring. In `mceliece348864` parameter set, the input size is $12 \times 64 = 768$ bits. A 64-coefficient polynomial over the quotient ring $GF_{2^{12}} := GF_2[x]/\langle g \rangle$, where $g(z) = z^{12} + z^3 + 1$, is repeatedly written into the sum of two polynomials $f(x) = f^0(x^2 + x) + xf^1(x^2 + x)$. This process iterates for 5 times, each time producing two half-sized polynomials, producing 32 linear polynomials in the end.

How can we rewrite a polynomial in terms of $(x^2 + x)$ efficiently? The textbook method is to apply long divisions repeatedly to lower down quotient's degree, and concludes at degree 1, collecting all the remainders back up. McBits does this cleverly by dividing with $(x^2 + x)^{2^k}$ with some suitable k such that the quotient and the remainder are having the same degree, and recursively process the quotients and remainders in parallel.* Despite the large exponents, a divisor always end up sparsely, as in $(x^2+x)^{2^k} = x^{2^{k+1}} + x^{2^k}$.

Furthermore, McBits stores the coefficients in a *bit-sliced* form. To store *bit-sliced* coefficients is to arrange bits in the way that each bit of all coefficients are gathered at the same limb. The idea is similar to the row-major versus column-major order when storing a matrix. Transposing the representation allows one to do bitwise XORs for all coefficients simultaneously, practicing a technique called SWAR (SIMD within a register).

At the end of each iteration, a twisting divides every term with the value of last basis, to keep the invariant that the sequence of basis having 1 in its end. The scaling is a vectorized, element-wise multiplications to a predefined table, and then modulo g to

* Refer to problem 14 of Section 4.4 of [13], pp. 328 for the idea.

doi:10.6342/NTU202300977

keep the elements the canonical representation in $GF_{2^{12}}$, so the next stage can be again carried out in XORs without the need of dealing with overflows.

The outer-most loop, indexed with `j`, denotes the current stage. The middle loop, indexed with `i`, iterates over all bits of the variables. Because the input variables are bit-sliced, each specific bits can be updated independently. The inner-most loop, indexed with `k`, controls the modulus for this single run. This includes the mask to use and the shift amount.

Divisions are done in-place by a pair of bitwise operations in parallel. For example, when taking a division by $(x^4 + x^2)$, it extracts higher bits of dividend, shift downwardly 2 bits, and then adds (XORs) back. Then continue to do so on lower bits. It resembles to what we usually calculate long divisions on paper, but done in parallel. For example, to compute $(x^{11} + x^9 + x^7 + x^4)$ divided by $(x^4 + x^2)$ in $GF_2[x]$, the computation looks like:

$$\begin{array}{lll} & \textit{quotient} & \textit{remainder} \\ (x^4 + x^2)(0) + x^{11} + x^9 + x^7 + x^4 & (00\ 00\ 00\ 00)_2 & (\underset{*_1}{\underline{10}}\ 10\ \underset{*_1}{\underline{10}}\ 01\ 00\ 00)_2 \\ = (x^4 + x^2)(x^7 + x^3) + x^5 + x^4 & (\underset{*_1}{\underline{10}}\ 00\ \underset{*_1}{\underline{10}}\ 00)_2 & (00\ \underset{*_2}{\underline{00}}\ 00\ \underset{*_2}{\underline{11}}\ 00\ 00)_2 \\ = (x^4 + x^2)(x^7 + x^3 + x + 1) + x^3 + x^2 & (10\ \underset{*_2}{\underline{00}}\ 10\ \underset{*_2}{\underline{11}})_2 & (00\ 00\ \underline{00}\ 00\ \underline{11}\ 00)_2 \end{array} \tag{5.4}$$

Note that the in-place division in Equation (5.4) cannot be done at once, otherwise some of the limbs would be read from and written to at the same time. To yield the correct result, at least two passes are required, arranged in an alternated pattern, as marked q_1 and q_2. At later stages, smaller polynomials are placed adjacent to each other, and a pair of suitable masks is applied to compute quotients and remainders. At the end of each stage is the twisting operation, commented with "scaling" in the original code.

doi:10.6342/NTU202300977

5.4 Extraction of assembly code

In order to extract a trace of the library function `radix_conversions`, a driver code snippet is required for Listing 5.1:

Listing 5.2: The driver code for `rad.c`

```
int main() {
    uint64_t inp[GFBITS] = {};
    radix_conversions(inp);
}
```

To simplify, we first substitute the subroutine of multi-limb multiplication (saved as `asm.S`) to a stub function that contains a single `ret`, to keep it from the trace. Once checked, its post-condition can be pasted into the CryptoLine source, and we claim that the program structure is unchanged. This is no longer true if the compiler chooses to inline the function, but is fine for our verification process.

Compiling it will result in an executable `rad` containing the subroutine in question*. The executable is then passed to `itrace.py` in order to be inspected. Command lines are as follows:

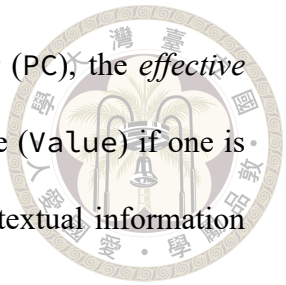
Listing 5.3: Commands to compile and invoke `itrace.py`

```
gcc -O2 -g -fomit-frame-pointer -fPIC -fPIE rad.c asm.S -o rad
python script/itrace.py rad radix_conversions trace.gas
```

`itrace` script starts a GDB session and communicates with it interactively. Under the hood, it steps into the program, record the instructions it goes through, along with

* The C and assembly source code are compiled with gcc 10.1.2 under Debian 11 (Bullseye) with suitable compiler flags from SUPERCOP.

doi:10.6342/NTU202300977



information gathered from its context, such as the *program counter* (`PC`), the *effective address* (`EA`) if the source is a pointer indirection, or the scalar value (`Value`) if one is involved. This results the following trace in GAS format, with contextual information annotated as comments.

The code listing is excerpted and explained below to outline a typical workflow of a verification task. Code is reformatted for display purpose, and annotated comments related to the program counter are stripped for brevity. The subroutine is small enough that a register serves exactly one purpose and no registers are spilled to memory, which simplies things a lot.

The function begins with its prologue to setup the call frame and constants are initialized. The compiler uses `rep` to transfer a chunk of data. First we can see that the stack pointer `%rsp` is moved backward in order to save these constants in stack memory. Our convention is that memory locations are directly mapped to variable names in `L0xnnn` form.

Listing 5.4: The assembly program of radix conversions

```
1  radix_conversions:
2  # %rdi = 0x7fffffffdaa0
3  # %rsi = 0x7fffffffdc08
4  # %rdx = 0x7fffffffdc18
5  # %rcx = 0x7ffff7fa3718
6  # %r8  = 0x0
7  # %r9  = 0x7ffff7fe21b0
8  #! -> SP = 0x7fffffffda98
9  push   %r15                                    #! EA = L0x7fffffffda90
10 lea    0xdf7(%rip),%rsi    # 0x555555556020
11 mov    %rdi,%r15
12 mov    $0x3c,%ecx
```


doi:10.6342/NTU202300977

```
13  movabs $0x8888888888888888,%rax
14  push   %r14                                  #! EA = L0x7fffffffda88
15  push   %r13                                  #! EA = L0x7fffffffda80
16  lea    0x60(%r15),%r13
17  push   %r12                                  #! EA = L0x7fffffffda78
18  movabs $0xffff00000000,%r12
19  push   %rbp                                  #! EA = L0x7fffffffda70
20  xor    %ebp,%ebp
21  push   %rbx                                  #! EA = L0x7fffffffda68
22  sub    $0x238,%rsp
23  mov    %rax,(%rsp)                           #! EA = L0x7fffffffd830
24  lea    0x50(%rsp),%rbx
25  mov    %rsp,%r14
26  movabs $0x4444444444444444,%rax
27  mov    %rax,0x8(%rsp)                        #! EA = L0x7fffffffd838
28  mov    %rbx,%rdi
29  movabs $0xc0c0c0c0c0c0c0c0,%rax
30
31  rep movsq %ds:(%rsi),%es:(%rdi)
32                          #! EA = L0x555555556020; Value = 0xf3cfc030fc30f003
33  # ... [repeated 58 times more] ...
34  rep movsq %ds:(%rsi),%es:(%rdi)
35                          #! EA = L0x5555555561f8; Value = 0x0000000000000000
36
37  mov    %rax,0x10(%rsp)                       #! EA = L0x7fffffffd840
38  movabs $0x3030303030303030,%rax
39  mov    %rax,0x18(%rsp)                       #! EA = L0x7fffffffd848
40  movabs $0xf000f000f000f000,%rax
41  mov    %rax,0x20(%rsp)                       #! EA = L0x7fffffffd850
42  movabs $0xf000f000f000f00,%rax
43  mov    %rax,0x28(%rsp)                       #! EA = L0x7fffffffd858
44  movabs $0xff000000ff000000,%rax
45  mov    %rax,0x30(%rsp)                       #! EA = L0x7fffffffd860
46  movabs $0xff000000ff0000,%rax
47  mov    %rax,0x38(%rsp)                       #! EA = L0x7fffffffd868
```


doi:10.6342/NTU202300977

```
48  movabs $0xffff000000000000,%rax
49  mov    %rax,0x40(%rsp)                          #! EA = L0x7fffffffd870
```

It follows the main body of the subroutine. It's a nested triple loop, but we can only see its unrolled form from the trace. Nevertheless, since it is a constant-time subroutine, the only possible source of branching is the for-loops. For example, `%ebp` is compared against `%esi`, which is a constant 4. This strongly suggests that it is what `j` compiles to.

```
50  mov    %r15,%r9
51  mov    $0x1,%r8d
52  nopl   (%rax)                                   #! EA = L0x-1000000000000
53  mov    (%r9),%rax
54                          #! EA = L0x7fffffffdaa0; Value = 0x0000000000000002
55  mov    %r14,%rdi
56  mov    %r12,%rdx
57  mov    $0x4,%esi
58  mov    %esi,%ecx
59  mov    0x40(%rdi),%r10
60                          #! EA = L0x7fffffffd870; Value = 0xffff000000000000
61  mov    %r8d,%r11d
62  sub    $0x1,%esi
63  shl    %cl,%r11d
64  sub    $0x10,%rdi
65  mov    %r11d,%ecx
66  and    %rax,%r10
67  shr    %cl,%r10
68  xor    %r10,%rax
69  and    %rax,%rdx
70  shr    %cl,%rdx
71  xor    %rdx,%rax
72  cmp    %ebp,%esi
73  #jl     0x555555555330 <radix_conversions+272>
```


doi:10.6342/NTU202300977

Similarly, it can be deduced that `esi` is holding the value for the innermost loop variable `k`. And the loop body can be clearly seen as above.

To find out what the C code

```
in[i] ^= (in[i] & mask[k][0]) >> (1 << k);
```

maps to in assembly, prior knowledge of assembly language may be helpful. Other than that, the reader can make a few guesses to tell the full story. As a demonstration, breaking the statement down into more basic operations and matching them in assembly code, result in the following:

1. `tmp0 = in[i] & mask[k][0]`

   ⟶ `and` (bitwise-AND) `%r10` and `%rax`

2. `tmp1 = 1 << k`

   ⟶ `shl` (logical shift left) with base `%r11d`, amount `%cl`

3. `tmp2 = tmp0 >> tmp1`

   ⟶ `shr` (logical shift right) with base `%r10`, amount `%cl`

4. `in[i] = in[i] ^ tmp2`

   ⟶ `xor` (bitwise-XOR) `%rax` and `%r10`

The analysis reveals that the input is stored in `%r10`, and the mask is stored in `%rax`. The input is bitwise-and with the mask, shifted left with amount `%cl`, then shifted right, and then XOR'ed with the input itself, all done in-place.

```
74  mov     0x48(%rdi),%rdx
75                          #! EA = L0x7fffffffd868; Value = 0x00ff000000ff0000
76  #jmp    0x5555555552fe <radix_conversions+222>
77  mov     %esi,%ecx
78  mov     0x40(%rdi),%r10
```


doi:10.6342/NTU202300977

```
                        #! EA = L0x7fffffffd860; Value = 0xff000000ff000000
mov     %r8d,%r11d
sub     $0x1,%esi
shl     %cl,%r11d
sub     $0x10,%rdi
mov     %r11d,%ecx
and     %rax,%r10
shr     %cl,%r10
xor     %r10,%rax
and     %rax,%rdx
shr     %cl,%rdx
xor     %rdx,%rax
cmp     %ebp,%esi
#jl      0x555555555330 <radix_conversions+272>
```

Then, the outer two levels of loop goes on, until it arrives at a function call to `_PQCLEAN_MCELIECE348864_AVX_vec_mul_asm`. `%rbp` appears as a new register, but `%ebp` is just the lower 32 bits of it. At the scope of this subroutine, they basically point to the same value.

```
mov     0x48(%rdi),%rdx
                          #! EA = L0x7fffffffd858; Value = 0
                            x0f000f000f000f00
#jmp     0x5555555552fe <radix_conversions+222>

# ... [omitted] ...

cmp     %r9,%r13
#jne     0x5555555552f0 <radix_conversions+208>
lea     0x0(%rbp,%rbp,2),%rdx
mov     %r15,%rsi
mov     %r15,%rdi
add     $0x1,%rbp
```


doi:10.6342/NTU202300977

```
105  shl     $0x5,%rdx
106  add     %rbx,%rdx
107  #call    0x555555555380 <_PQCLEAN_MCELIECE348864_AVX_vec_mul_asm>
108  #! -> SP = 0x7fffffffd828
109  #! <- SP = 0x7fffffffd828
110  #ret
111  cmp     $0x5,%rbp
112  #jne     0x5555555552e4 <radix_conversions+196>
113  mov     %r15,%r9
114  mov     $0x1,%r8d
115
116  # ... [omitted] ...
```

Having understood the trace, a verifier can make decisions on how to model in CryptoLine. Here are some general ideas:

- **Variable initialization.** Various constants are pushed on to the stack at the beginning of code. By performance reasons, the compiler may initiate some of the constant in the registers directly, and/or use some specialized instructions (like `rep movsq`), additionally interleaving them with the function prologue. These optimizations make them trickier to be recognized or translated. Due to the efforts we make here are hardly reused later, we suggest using our own code for initialization.
- **The loop variables.** `itrace` unrolls all loops by its nature, but it leaves comments at each jump to help us reconstruct the loop structure. Furthermore, these comments are hooks for other scripts to parse and instrument extra code or specifications. It is thus worth locating the loop variables. This is a non-trivial task, as compiler often optimize these induction variables out. Additionally, since the loops are fully

doi:10.6342/NTU202300977

unrolled, all branching instructions `cmp` are futile and thus need to be replaced with `nops`.

- **Shift amounts.** Here we need to keep eye on platform-specific rules of registers (in this case x86_64). Take the register `rax` for example: `eax` is the lower 32 bits of `rax`, and setting `eax` effectively clears the upper 32 bits of `rax`. We keep these distinct names and use value-perserving casts (`vpc`) to synchronize between them.
- **Input/output variables.** We keep track of the names and decide how to model them with VCL vectors. In our case, each register packs bits of respective position from each and every coefficients. Hence, it is the most natural to model them as single-bit vectors.
- Other registers that are irrelevant to the core algorithm, e.g., address indirections. We can comment out instructions containing them, but keeping them in the source code does not hurt performance. CryptoLine will detect and slice out those unused variables when specifying the argument `-slicing`.

For each scalar variable (such as `rdi`), we write a translation rule `%rdi = %%rdi` in the beginning of the GAS file, where `%%` is the escaped form of a `%` symbol. For each vector variable (such as `r10`), an additional `%` is prepended to its name, so the rule becomes `%r10 = %%%r10`. Instructions are also translated, but their rules are matched against regexes, which are slightly more complex.

Translation rules are written as comments in the GAS file while we keep inspecting the output of `to_zdsl.py`, until it is a well-formed Cryptoline DSL. Note that CryptoLine will not accept this file at this moment, as some instructions need special treatments, which are filled out after.

doi:10.6342/NTU202300977

5.5 Modeling the bit-shiftings

In order to relate the result of additive FFT with these variables, which involves some nonlinear calculations, we had better use algebraic properties whenever possible. Immediately we meet a challenge that `shl` and `shr` are not translatable to algebraic relations:

```
shl r11d cl
```

To make things worse, CryptoLine only support constant shift amount or exponents at time of writing, but the index `k` (`cl` in above) is not.

To deal with variable shift amounts, several attempts are made. Observe that `k` only takes one of the values $\{0, 1, 2, 3, 4\}$, this means that `tmp1` only takes one of the values $\{1, 2, 4, 8, 16\}$. We anticipate a switch-case like structure in the code.

The first attempt is made around algebraic specifications. Each algebraic specification translates into ideal membership problems with each condition induces a ring ideal generated by some polynomial. Finite conjunctions of specifications can be solved simultaneously, while there is no such correspondence for a finite disjunction.

However, we can still derive an algebraic specification in question using the logic equivalence $A \to B \equiv A \vee \neg B$. Observe that a specification of the form `A_1 * A_2 * ⋯ * A_n = 0` holds if and only if one of `A_1`, `A_2`, …, or `A_n` is zero.

Based on these building blocks, the original value of `r11d` is saved to `r11d_0` at first and all possible `i` is enumerated, writing the condition of the form that A is `tmp1 = 1 << i` and B is `k = i`. This yields:

doi:10.6342/NTU202300977

```
assert true && and [cl >= 0@8, cl <= 4@8];
mov r11d_0 r11d;
assume (r11d - r11d_0     ) *
       (cl - 1) * (cl - 2) * (cl - 3) * (cl - 4) = 0 && true;
assume (r11d - r11d_0 *  2) *
       (cl - 0) * (cl - 2) * (cl - 3) * (cl - 4) = 0 && true;
assume (r11d - r11d_0 *  4) *
       (cl - 1) * (cl - 0) * (cl - 3) * (cl - 4) = 0 && true;
assume (r11d - r11d_0 *  8) *
       (cl - 1) * (cl - 2) * (cl - 0) * (cl - 4) = 0 && true;
assume (r11d - r11d_0 * 16) *
       (cl - 1) * (cl - 2) * (cl - 3) * (cl - 0) = 0 && true;
# r11d is now left-shifted by cl
```

For instance, the first `assume` holds either `r11d - r11d_0 = 0` or `cl` equals to anything other than `0`. This solution seems appropriate, but does not scale well due to the condition, when converted to a polynomial to CAS, being so huge that it freezes in our test environment even when there is only one iteration (60 `shl`s in total). This solution might still be useful in practice when there are only a couple of occurrences.

The workaround is to emulate the nested for-loop and instrument the code with assertions. A Python script is run that scan through the entire code, bookkeeping the values of loop variables. At each use of `k`, an `assert` is instrumented to prove with an SMT solver that it is equal to the value with `assert`'s, and then transfer it to CAS with `assume`'s. The same approach is applied to `shr` instructions. The obvious disadvantage to this solution is that it requires us to maintain a separate script to emulate the loop. To ensure the soundness, i.e., that the emulation matches the implementation, it is crucial that we guard these variable with additional `assert`'s. Here we can see how CryptoLine's combination of two techniques yield the best of both worlds.

doi:10.6342/NTU202300977

5.6 Generating the specifications

We place two `cut` statements near the end of each stage. `cut` is a statement that works like a checkpoint to speed up our verification process. When a `cut` statement is reached by CryptoLine, it splits up the program into two smaller ones: the former with its specification as the post-condition, the latter with its specification as pre-condition. This way, the latter program can replace some information it learned previously with a summarized pre-condition, effectively reducing the magnitude of the problem ([12], pp. 12–13).

The two cuts are placed before and after the twisting process, respectively. One is to verify the radix conversion successfully yield two smaller polynomials, and the other is right after the scaling multiplications to verify the factors used in the scaling process matches the mathematical derivation. Only the first layer is demonstrated as below, but the idea can be generalized to all layers.

The first part looks like below. All coefficients are unpacked to bits from the memory addresses `0x7fffffffdaa0` through `0x7fffffffdaf8` to names `out0` through `out11`, and then use a `ghost` instruction to reconstruct the 12-bit coefficients of the output polynomial. A `ghost` instruction declares special variables that are only visible to specifications. The assert can be written as follows:

$$\texttt{input\_poly} \equiv \sum_{i=0}^{31} (\texttt{cvrted0}_{2i} + x \cdot \texttt{cvrted0}_{2i+1})(x^2+x)^i \pmod 2, \tag{5.5}$$

where

$$\texttt{cvrted0}_i := \sum_{j=0}^{11} (\texttt{out}i)_j z^j \quad \text{for } i = [0..63]. \tag{5.6}$$

doi:10.6342/NTU202300977

The cut statement is written like this in Cryptoline.

```
mov [out0_0, out0_1, ..., out0_63] %L0x7fffffffdaa0;
mov [out1_0, out1_1, ..., out1_63] %L0x7fffffffdaa8;
...
mov [out11_0, out11_1, ..., out11_63] %L0x7fffffffdaf8;

ghost cvrted0_0@uint12: cvrted0_0 = out0_0 * z**0 + out1_0 * z**1 + out2_0
   * z**2 + ... + out11_0 * z**11 && true;
ghost cvrted0_1@uint12: cvrted0_1 = out0_1 * z**0 + out1_1 * z**1 + out2_1
   * z**2 + ... + out11_1 * z**11 && true;
...
ghost cvrted0_63@uint12: cvrted0_63 = out0_63 * z**0 + out1_63 * z**1 +
  out2_63 * z**2 + ... + out11_63 * z**11 && true;

ecut eqmod input_poly (
  (cvrted0_0 + x * cvrted0_1) * (x ** 2 + x) ** 0 +
  (cvrted0_2 + x * cvrted0_3) * (x ** 2 + x) ** 1 +
  ... +
  (cvrted0_62 + x * cvrted0_63) * (x ** 2 + x) ** 31
) 2;
#call   0x555555555380 <_PQCLEAN_MCELIECE348864_AVX_vec_mul_asm>#! 0
  x555555555352 = 0x555555555352; ...
```

The control flow reaches the function `_PQCLEAN_MCELIECE348864_AVX_vec_mul_asm`, which is supposed to be verified elsewhere. To proceed, it is required to convince CryptoLine that it is verified by simulating its behavior with `assume` statements.

Define variables `res0_i` to hold the expected result of multiplications. That is, each `cvrted0_i` gets multiplied with the corresponding scaling factors, then take modulo of $g = z^{12} + z^3 + 1$. For example, at the first iteration, the scaling factors are powers of

doi:10.6342/NTU202300977

$z^2 + z$, it can be written as:

$$\begin{aligned}
\texttt{res0}_i &\equiv \texttt{cvrted0}_i \times (z^2+z)^{\lfloor i/2 \rfloor} && (\mathrm{mod}\ (2, g)) \quad \text{for } i = [0..63]; \text{ or} \\
\texttt{res0}_i &\equiv \texttt{cvrted0}_i \times \underbrace{\left((z^2+z)^{\lfloor i/2 \rfloor} \bmod g\right)}_{\text{precomputed}} && (\mathrm{mod}\ (2, g)) \quad \text{for } i = [0..63]
\end{aligned} \tag{5.7}$$

Because the factors were constants, their polynomial representations were prepared with a SageMath* script, and were plugged into the code below, as indicated by the comments `(* 0 0 *)` through `(* 0 63 *)`. In the modular equation, `eqmod` accepts multiple moduli to mark our intention that the assumption is both on modulo 2 and g (held by the variable `modulus` in the code). In CryptoLine's ring of choice, the order does not affect the result.

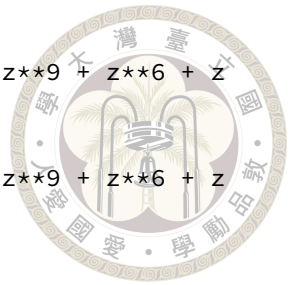
```
nondet res0_0@bit; nondet res0_1@bit; nondet res0_2@bit; nondet res0_3@bit
  ;
...
nondet res0_60@bit; nondet res0_61@bit; nondet res0_62@bit; nondet
  res0_63@bit;

assume and [
  eqmod res0_0 (cvrted0_0 * ((* 0  0 *) 1)) [2, modulus],
  eqmod res0_1 (cvrted0_1 * ((* 0  1 *) 1)) [2, modulus],
  eqmod res0_2 (cvrted0_2 * ((* 0  2 *) z**2 + z)) [2, modulus],
  eqmod res0_3 (cvrted0_3 * ((* 0  3 *) z**2 + z)) [2, modulus],
  eqmod res0_4 (cvrted0_4 * ((* 0  4 *) z**4 + z**2)) [2, modulus],
  eqmod res0_5 (cvrted0_5 * ((* 0  5 *) z**4 + z**2)) [2, modulus],
  ...
  eqmod res0_60 (cvrted0_60 * ((* 0 60 *) z**11 + z**9 + z**7 + z**6 + z
    **5 + z**2 + z + 1)) [2, modulus],
  eqmod res0_61 (cvrted0_61 * ((* 0 61 *) z**11 + z**9 + z**7 + z**6 + z
    **5 + z**2 + z + 1)) [2, modulus],
```

* SageMath (https://www.sagemath.org/) is an open-source mathematical software based upon Python, licensed under GPLv2. Its source code is available at https://github.com/sagemath/sage.

doi:10.6342/NTU202300977

```
    eqmod res0_62 (cvrted0_62 * ((* 0 62 *) z**11 + z**10 + z**9 + z**6 + z
      **3 + 1)) [2, modulus],
    eqmod res0_63 (cvrted0_63 * ((* 0 63 *) z**11 + z**10 + z**9 + z**6 + z
      **3 + 1)) [2, modulus]
  ] && true;
```



The second cut is placed to assert that the coefficients are split into two twisted polynomials, as expressed in the following equations,

$$f^0(y) = \sum_{i=0}^{31} \texttt{cvrted0}_{2i}\, y^i \quad \text{and} \quad f^1(y) = \sum_{i=0}^{31} \texttt{cvrted0}_{2i+1}\, y^i \quad \text{with a free variable } y \tag{5.8}$$

Intuitively, we can choose the original variable `x` as the y to connect the two sequences of results, but here is where it gets tricky. Indeed, it is a indirect substitution to say $x_1 = x^2 + x$, but declaring a variable makes the intention more explicit to CryptoLine, which is essential in this case for a CAS to unfurl the hidden structure.* Finally, let `x1` become the new `x` before exiting the loop body.

```
nondet x1;

ecut and [
  eqmod (
    (cvrted0_0 + x * cvrted0_1) * ((z ** 2 + z) * x1) ** 0 +
    (cvrted0_2 + x * cvrted0_3) * ((z ** 2 + z) * x1) ** 1 +
    ...
    (cvrted0_60 + x * cvrted0_61) * ((z ** 2 + z) * x1) ** 30 +
    (cvrted0_62 + x * cvrted0_63) * ((z ** 2 + z) * x1) ** 31
  ) (
    (res0_0 + x * res0_1) * x1 ** 0 +
    (res0_2 + x * res0_3) * x1 ** 1 +
```

* Technically, it adds a polynomial $x_1 - (x^2 + x)$ to the set of ideals. From the experiments, the CAS is observed to reduce the result term drastically faster.

doi:10.6342/NTU202300977

```
    ...
    (res0_60 + x * res0_61) * x1 ** 30 +
    (res0_62 + x * res0_63) * x1 ** 31
  ) [2, modulus]
];

mov x x1;
```

After the cut, results of our simulated computations are put back to the variables: Use `assumes` to extract individual bits, combine and rearrange them, and then assign them to memory variables. Another two ghost variables serving as the input polynomials to the next level.

$$\texttt{inp1}_0 := f^0(x_1) \quad \text{and} \quad \texttt{inp1}_1 := f^1(x_1)$$

```
ghost inp1_0@uint12: inp1_0 =
  res0_0 * x**0 + res0_2 * x**1 + res0_4 * x**2 + res0_6 * x**3 +
  ...
  res0_56 * x**28 + res0_58 * x**29 + res0_60 * x**30 + res0_62 * x**31
&& true;
ghost inp1_1@uint12: inp1_1 =
  res0_1 * x**0 + res0_3 * x**1 + res0_5 * x**2 + res0_7 * x**3 +
  ...
  res0_57 * x**28 + res0_59 * x**29 + res0_61 * x**30 + res0_63 * x**31
&& true;

nondet rb0_0@bit; nondet rb0_1@bit; nondet rb0_2@bit; ... nondet
  rb0_63@bit;
nondet rb1_0@bit; nondet rb1_1@bit; nondet rb1_2@bit; ... nondet
  rb1_63@bit;
...
nondet rb11_0@bit; nondet rb11_1@bit; nondet rb11_2@bit; ... nondet
  rb11_63@bit;
```


doi:10.6342/NTU202300977

```
16
17  assume and [
18    res0_0 = rb0_0 * z**0 + rb1_0 * z**1 + rb2_0 * z**2 + rb3_0 * z**3 +
        rb4_0 * z**4 + rb5_0 * z**5 + rb6_0 * z**6 + rb7_0 * z**7 + rb8_0 * z
        **8 + rb9_0 * z**9 + rb10_0 * z**10 + rb11_0 * z**11,
19    res0_1 = rb0_1 * z**0 + rb1_1 * z**1 + ... + rb11_1 * z**11,
20    ...
21    res0_63 = rb0_63 * z**0 + rb1_63 * z**1 + ... + rb11_63 * z**11
22  ] && true;
23
24  mov %L0x7fffffffdaa0 [rb0_0, rb0_1, ..., rb0_63];
25  mov %L0x7fffffffdaa8 [rb1_0, rb1_1, ..., rb1_63];
26  ...
27  mov %L0x7fffffffdaf8 [rb11_0, rb11_1, ..., rb11_63];
```

The whole subroutine makes 5 iterations of radix conversion and scaling in alternative. Plus a dummy condition that is use to *forget* the previous cut condition, there are 11 cuts in total. They are referred by index from 0 to 10.

5.6.1 The post-condition

From the abovementioned 11 cuts, the following post-condition is proposed. Observe that the input polynomial f is of degree 63, it can be uniquely determined from the evaluations on 64 points. By virtue of this fact, it suffices to pick 64 points and only check the evaluations on them. The most straightforward choice is all the subset sums of basis $\{z^5, z^4, z^3, z^2, z, 1\}$ itself, that is,

$$\{m_5 z^5 + m_4 z^4 + \cdots + m_0 : m_5, m_4, \cdots, m_0 \in \{0, 1\}\}.$$

doi:10.6342/NTU202300977

With every operation being linear and reversible, we can express every evaluation as a linear combination of the output, effectively undoing the inverse of radix conversions and twistings.

We use the following convention: For a polynomial f_i (i can be empty), the notation f_{i0} and f_{i1} are the resulting polynomials from the radix conversion, and the notation ${f_i}^t$ *twists* f_i, evaluating it at a new point that the last element in the basis is cancelled out. Some basic facts are illustrated as the relationship between f and the two sub-polynomials f_0 and f_1 as follows.

Unroll one layer for $f(x) = f_0^t(\beta(x^2+x)) + x f_1^t(\beta(x^2+x))$ supposing the bits after the twist are $\sum_{i=0}^{63} 2^i d_i = d_0 + 2d_1 + 2^2 d_2 + \cdots + 2^{63} d_{63}$ and scaling factor β, we have *indeterminate* results

$$\begin{aligned} f_0^t(y) &= d_0 + d_2 y + d_4 y^2 + \cdots + d_{62} y^{31}; \quad \text{and} \\ f_1^t(y) &= d_1 + d_3 y + d_5 y^2 + \cdots + d_{63} y^{31}. \end{aligned} \tag{5.9}$$

Let us define the *final* output bits of radix conversion after the last layer $\sum_{i=0}^{63} 2^i c_i$. We can see that $f(0)$ equals to the constant coefficient of f_0, which cascades to the constant term of the output; $f(1)$ equals to the sum of f_0 and f_1 evaluated at 0, which is the sum of constant terms of those functions, or the outputs. That is,

$$\begin{aligned} f(0) &= f_0(0) &&= d_0, \qquad \text{and} \\ f(1) &= f_0(0) + f_1(0) = d_0 + d_1. \end{aligned} \tag{5.10}$$

Notice that these terms that are not affected by scaling (i.e., the scaling factor for constant terms are always 1), so d_0 and d_1 are indeed equal to the output c_0 and c_1, respectively. By the end of the 5 layers, all 2^5 polynomials are linear terms (as explained in Section 5.1), and so all c_i's are going to be resolved.

doi:10.6342/NTU202300977

Listing 5.5: Sample of derived post-conditions in CryptoLine

```
assert and [
  # f(0) and f(1) from Equation (5.10)
  eqmod input_poly (c0)
        [2, modulus, x0, x1, x2, x3, x4, x5],
  eqmod input_poly (c0 + c1)
        [2, modulus, x0 - 1, x1, x2, x3, x4, x5],
  # f(z) from Equation (5.11)
  eqmod input_poly (c0 + c2) + z * (c1 + c3)
        [2, modulus, x0 - z, x1 - 1, x2, x3, x4, x5]
] prove with [precondition, cuts [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]] && true;
```

These equations shall be derived from all cuts. The `precondition` is to refer to the definition of the modulus. Each specification collected with `prove with` is inserted as an `assume` right after the pre-condition of the latter half of the cut. They are effectively conjunctive to the cut-supplied pre-conditions.

The loop can be unrolled once more to see how $f(z)$ can be expressed by c_i. Now that z is the second-last element of the basis, the parameter drops to exactly 1 after one stage. The following expression can be derived:

$$\begin{aligned} f(z) &= f_0(z^2+z) + z f_1(z^2+z) \\ &= f_0^t(1) + z f_1^t(1) = c_0 + c_2 + z(c_1 + c_3), \end{aligned} \tag{5.11}$$

with

$$\begin{aligned} f_0^t(x) &= (d_0 + d_2 x) + (d_4 + d_6 x)(x^2+x) + \cdots + (d_{60} + d_{62})(x^2+x)^{15}, \quad \text{and} \\ f_1^t(x) &= (d_1 + d_3 x) + (d_5 + d_7 x)(x^2+x) + \cdots + (d_{61} + d_{63})(x^2+x)^{15}. \end{aligned} \tag{5.12}$$

These equations are written into CryptoLine specification as shown in Listing 5.5. Following the pattern, it is clear that an evaluation on any basis element can always

doi:10.6342/NTU202300977

be expressed as a linear combination of c_i, whose coefficients are revealed by some backtracking.

All 64 elements can be computed from linear combinations of the basis elements. Next, the mapping $x \mapsto x^2 + x$ is an homomorphism under characteristic 2 (in fact, an isomorphism).

It can be shown that the evaluation at a sum of any two subset elements a, b of the basis, is also linear.

$$\begin{aligned} f(a+b) &= f_0((a+b)^2 + a + b) + (a+b)f_1((a+b)^2 + a + b) \\ &= f_0(a^2 + a + b^2 + b) + (a+b)f_1(a^2 + a + b^2 + b) \\ &= f_0^t(a' + b') + (a+b)f_1^t(a' + b'), \end{aligned} \tag{5.13}$$

where a' and b' are the twisted elements of a and b respectively.

The following illustrates one example, say z^2+z, expressed as the linear combination of f_0 and f_1 and in turn expressed as of $f_{00}, f_{01}, f_{10}, f_{11}$, where the recursion stops.

$f(z^2 + z)$

{ radix conversion on f: $z^2 \mapsto z^4 + z^2$, $z \mapsto z^2 + z$ }

$= f_0(z^4 + z) + (z^2 + z)f_1(z^4 + z)$

{ twisting: dividing by $z^2 + z$ }

$= f_0^t\left(\dfrac{z^4 + z}{z^2 + z}\right) + (z^2 + z)f_1^t\left(\dfrac{z^4 + z}{z^2 + z}\right)$

$= f_0^t(\underline{z^2 + z} + \underline{1}) + (z^2 + z)f_1^t(\underline{z^2 + z} + \underline{1})$

{ radix conversion on f_0 and f_1: $z^2 + z \mapsto z^4 + z$, $1 \mapsto 0$ }

$= f_{00}(\underline{z^4} + \underline{z}) + (z^2 + z + 1)f_{01}(\underline{z^4} + \underline{z})$

doi:10.6342/NTU202300977

$$+ (z^2+z)[f_{10}(\underline{z^4+z}) + (z^2+z+1)f_{11}(\underline{z^4+z})]$$

{ twisting: dividing by z^4+z }

$$= f_{00}^t(1) + (z^2+z+1)f_{01}^t(1) + (z^2+z)[f_{10}^t(1) + (z^2+z+1)f_{11}^t(1)]$$

{ apply Equation (5.10) on base case $x = 1$ }

$$= c_0 + c_4 + (z^2+z+1)(c_2+c_6) + (z^2+z)[(c_1+c_5) + (z^2+z+1)(c_3+c_7)].$$

In the former twisting process, we do not need to know $(z^2+z)^{-1}$ to explicitly compute $(z^4+z)/(z^2+z)$. The two basis simply guide us through the conversion. By computing all scaling factors beforehand, we can work out the accurate algebraic specifications. All it remains is to make CryptoLine prove them.

5.6.2 Implemention details

The models are evaluated and benchmarked on a server with an Intel Xeon Gold CPU with 48 × 2.3GHz cores and 810 GB memory. The operation system is Ubuntu 22.04 with dependencies Singular version 4.2.1 and Boolector version 3.2.0.

The above specifications are developed for `mceliece348864`, but they are reusable across different parameter sets (the chosen irreducible polynomial and the size of the base field) of Classic McEliece, like `mceliece460896`, `mceliece6688128`, `mceliece6960119`, `mceliece8192128` [18].

To accelerate the development in the initial experimenting phase, a degenerate case is developed taking only the LSB of each coefficient, and fills other bits with zeros. This effectively restricts the domain of coefficients to GF_2 for faster feedback.

doi:10.6342/NTU202300977

5.7 Verification result

The overall verification result is passing. We outline the aspects of verification and their benchmarks below:

Safety: The radix conversion routine generates a total of ~24k (23993) safety conditions. With `-isafety` enabled and a 24-job worker pool, it takes ~3 hr to fully pass all the checks.

Range properties: All range assertions are merely doing the simulations. They are trivial to verify, but due to the numbers and appearances scattering all over the subroutine, it still takes about 251.7 ± 2.1 seconds to process.

Algebraic properties: The full model takes roughly 8 minutes (468.5 ± 25.8 sec) to verify. If the programmer only want to do a quick check over the whole implementation, it only takes about half a minute (24.82 sec) verifying LSBs. It is surprisingly fast compared to using traditional SMT model checkers.

The detailed time breakdown is shown in Table 5.1. The times are reported from the debug trace produced by CryptoLine. For full runs, the reported durations are measured from complete invocations of CryptoLine, and thus cover preprocessing like parsing DSL sources or rewriting expressions, etc. For individual cuts, the cut in interest is supplied from command-line arguments, and only the time of the relevant parts are reported.

It can be observed that the very first (the zeroth) radix conversion contributes to the most of algebraic verification time if the post-condition is ignored. We can tell that modelling divisions on a higher-degree polynomial is time-consuming than two

doi:10.6342/NTU202300977

half-degree divisions combined. The trend is opposite in cuts of twistings, where the transformations are straightforward, and the time spent is essentially proportional to the number of asserted conditions.

The model proves that the generated assembly correctly computes the radix conversion from the input polynomial. It also concludes with a post-condition from these verified cuts. This post-condition is meant to serve as the pre-condition to the subsequent butterfly operation.

Table 5.1: Running time of verifying the radix conversion subroutine

Task sampled with 5 runs each		Time (seconds) Average	STD
Full runs	Overall time	722.8	26.7
	Range properties	251.7	2.1
	Algebraic properties	468.5	25.8
Individual radix conversion cuts	Cut #0	181.2	0.35
	Cut #2	47.2	0.34
	Cut #4	26.7	0.30
	Cut #6	19.3	0.21
	Cut #8	12.6	0.25
Individual Twisting cuts	Cut #1	0.498	0.012
	Cut #3	0.836	0.039
	Cut #5	1.51	0.06
	Cut #7	2.83	0.12
	Cut #9	5.19	0.26
The post-condition only; Cut #11		302.7	22.8

doi:10.6342/NTU202300977

Chapter 6 Conclusion

In this thesis, I designed an extension to Cryptoline DSL, and used it to model both hand-written or compiler-generated assembly code. On top of the overloaded DSL instructions, I derived and wrote down the specifications, which were thoroughly checked by Vector CryptoLine. Various strategies were discussed to express the properties I seek to ensure and correspondence was drawn between the specification, the algorithm, and the underlying mathematical structure.

For a formal verification framework to be persuasive, the "proof" must be concise enough to be reasoned about, and the process should be reproducible within a reasonable amount of time and memory. In this regard, Vector CryptoLine has indeed made the verification of SIMD code more straightforward; but there are still rooms for improvement. First, better support for platform-specific semantics of registers could be added. In addition, now that Vector CryptoLine has a proof of concept, syntax on specifications could be extended so that terser specifications could also be written. Third, as an ongoing exploration, are there concise representations of highly-structured permutations, especially what used in NTT routines? Introducing such a concept to Vector CryptoLine will further simplify the proofs currently available.

As the famous idiom goes, "don't roll your own crypto", any oversight in code may turn into catastrophic security risks in reality. Users of cryptographic libraries

doi:10.6342/NTU202300977

are supposed to only care about the relations between the input and the output. As a verifier, however, one often stares at the algorithm and asks for a deeper understanding like “how can I accurately summarize the current state in a mathematical formula?” In order to formalize our thinking, we seek the finest automated verification tools, and Vector CryptoLine is just a modest improvement.

doi:10.6342/NTU202300977

References

[1] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. 2013. McBits: Fast constant-time code-based cryptography. In *Cryptographic hardware and embedded systems - CHES 2013*, Springer Berlin Heidelberg, Berlin, Heidelberg, 250–272.

[2] Eric Cheng. 2016. Binary analysis and symbolic execution with angr. PhD thesis. PhD thesis, The MITRE Corporation.

[3] Dong Pyo Chi, Jeong Woon Choi, Jeong San Kim, and Taewan Kim. 2015. Lattice based cryptography for beginners. *Cryptology ePrint Archive*.

[4] Tung Chou. McBits revisited. In *Cryptographic Hardware and Embedded Systems–CHES: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, Springer, 213–231.

[5] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. 2021. NTT multiplication for NTT-unfriendly rings: New speed records for saber and NTRU on cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 159–188.

[6] Stephen A Cook and Stål O Aanderaa. 1969. On the minimum computation time of functions. *Transactions of the American Mathematical Society* 142, 291–314.

doi:10.6342/NTU202300977

[7] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. 2018. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In *Progress in cryptology – AFRICACRYPT 2018*, Springer International Publishing, Cham, 282–305.

[8] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation (PLDI'19)*, ACM, 1133–1148. DOI: http://dx.doi.org/10.1145/3314221.3314601.

[9] Shuhong Gao and Todd Mateer. 2010. Additive fast fourier transforms over finite fields. *IEEE Transactions on Information Theory* 56, 12, 6265–6272.

[10] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. 1985. Gauss and the history of the fast fourier transform. *Archive for History of Exact Sciences* 34, 3, 265–277. DOI: http://dx.doi.org/10.1007/BF00348431.

[11] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2022. Verified NTT multiplications for NISTPQC KEM lattice finalists: Kyber, SABER, and NTRU. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 718–750.

[12] Ming-Hsien Tsai Jiaxiang Liu Xiaomu Shi and Bo-Yin Yang. 2022. CryptoLine: A tutorial. Retrieved May 29, 2023 from https://github.com/fmlab-iis/cryptoline/blob/master/doc/tutorial.pdf.

[13] M Donald MacLaren. 1970. *The art of computer programming. Volume 2: Seminumerical algorithms (donald e. knuth)*. SIAM.

doi:10.6342/NTU202300977

[14] Robert J McEliece. 1978. A public-key cryptosystem based on algebraic coding theory. *Coding Thv* 4244, 114–116.

[15] John M Pollard. 1971. The fast fourier transform in a finite field. *Mathematics of computation* 25, 114, 365–374.

[16] Léo Ducas Roberto Avanzi Joppe Bos and Damien Stehlé. 2019. CRYSTALS-kyber (version 2.0) – submission to round 2 of the NIST post-quantum project.

[17] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. 2004. A nanopass infrastructure for compiler education. In *Proceedings of the ninth ACM SIGPLAN international conference on functional programming* (ICFP '04), Association for Computing Machinery, New York, NY, USA, 201–212. DOI: http://dx.doi.org/10.1145/1016850.1016878.

[18] Harshdeep Singh. 2020. Code based cryptography: Classic McEliece. Retrieved May 22, 2023 from https://arxiv.org/abs/1907.12754.

[19] Andrei L Toom. 1963. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet mathematics doklady*, 714–716.

[20] VAMPIRE -Virtual Applications and Implementations Research Lab. 2022. SUPERCOP. Retrieved May 22, 2023 from https://bench.cr.yp.to/supercop.html.

[21] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 1789–1806.

doi:10.6342/NTU202300977