VIETNAM NATIONAL UNIVERSITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

CAPSTONE PROJECT REPORT

RESEARCH AND DEVELOPMENT OF A MEMORY MANAGEMENT MODULE BASED ON ZERO-KNOWLEDGE PROOFS

Major: COMPUTER ENGINEERING

Committee:	Committee 8 Computer Science	
Instructors:	Dr. Nguyen An Khuong	
	Mr. Tran Anh Dung	Orochi Network
	Mr. Pham Nhat Minh	Orochi Network
Reviewer:	Mr. Huynh Hoang Kha, MSc	
Student:	Trinh Cao Thang	2014550

Ho Chi Minh City, June 2024

463

ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC BÁCH KHOA

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

KHOA: KH & KT Máy tính
BỘ MÔN: KHMT

NHIỆM VỤ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP
Chú ý: Sinh viên phải dán tờ này vào trang nhất của bản thuyết trình

HỌ VÀ TÊN: **Trịnh Cao Thắng** MSSV: **2014550**
NGÀNH: **Kĩ thuật Máy tính** LỚP: **MT20KTTN**

1. Đầu đề luận văn/ đồ án tốt nghiệp: "NGHIÊN CỨU VÀ PHÁT TRIỂN MODULE QUẢN LÝ VÀ CHỨNG THỰC BỘ NHỚ DỰA TRÊN CHỨNG MINH KHÔNG ĐỂ LỘ TRI THỨC"

2. Nhiệm vụ (yêu cầu về nội dung và số liệu ban đầu):

- Đề xuất phương pháp tổ chức bộ nhớ phát triển từ định nghĩa gốc của bộ nhớ trong một chương trình RAM và hiện thực bằng ngôn ngữ lập trình Rust.
- Đề xuất sử dụng hai cơ chế cam kết để quản lý bộ lưu vết thực thi ghi lại toàn bộ hoạt động truy xuất bộ nhớ trong quá trình thực thi chương trình RAM và hiện thực một cơ chế cam kết dùng cơ chế cam kết đa thức KZG.
- Đề xuất một hệ ràng buộc dùng để hiện thực và chứng minh tính nhất quán của bộ nhớ trong quá trình thực thi chương trình RAM sử dụng hệ thống chứng minh Halo2.
- Tính toán thời gian thực thi của các giải thuật trong KZG, tính toán thời gian thực thi của prover và verifier trong mạch chứng minh tính nhất quán của bộ nhớ.
- Viết báo cáo đồ án tốt nghiệp.

3. Ngày giao nhiệm vụ: 30/01/2024

4. Ngày hoàn thành nhiệm vụ: 20/05/2024

5. Họ tên giảng viên hướng dẫn:	**Phần hướng dẫn:**
1) Nguyễn An Khương	- Hướng dẫn chung và giám sát thực hiện
2) Trần Anh Dũng	- Hướng dẫn kĩ thuật
3) Phạm Nhật Minh	- Hướng dẫn kĩ thuật

Nội dung và yêu cầu LVTN/ ĐATN đã được thông qua Bộ môn.

Ngày .... tháng .... năm ....

CHỦ NHIỆM BỘ MÔN
(Ký và ghi rõ họ tên)

GIẢNG VIÊN HƯỚNG DẪN CHÍNH
(Ký và ghi rõ họ tên)

[illegible]

Nguyễn An Khương

PHẦN DÀNH CHO KHOA, BỘ MÔN:
Người duyệt (chấm sơ bộ):
Đơn vị:
Ngày bảo vệ:
Điểm tổng kết:
Nơi lưu trữ LVTN/ĐATN:

463

TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KH & KT MÁY TÍNH

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

Ngày 17 tháng 5 năm 2024

PHIẾU ĐÁNH GIÁ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP

(Dành cho người hướng dẫn)

1. Họ và tên SV: **Trịnh Cao Thắng**
MSSV: **2014550** Ngành (chuyên ngành): **KHMT**
2. Đề tài: **"RESEARCH AND DEVELOPMENT OF A MEMORY MANAGEMENT MODULE BASED ON ZERO-KNOWLEDGE PROOFS"**
3. Họ tên người hướng dẫn: **Nguyễn An Khương, Trần Anh Dũng và Phạm Nhật Minh**
4. Tổng quát về bản thuyết minh:

Số trang: **100**	Số chương: **9**
Số bảng số liệu: **7**	Số hình vẽ: **16**
Số tài liệu tham khảo:	Phần mềm tính toán:
Hiện vật (sản phẩm)	

5. Những ưu điểm chính của LV/ ĐATN:
- The outcomes of this project can be compiled into a manuscript for future publication.
- Cao-Thang is a remarkable student known for his strong ability to study independently and conduct scholarly research.
-This project addresses the very challenging issue of ensuring consistent memory, which is a crucial aspect of developing reliable zero-knowledge proofs for RAM program execution.
- The research problem is not only very difficult but also carries significant importance for the creation of Zero-Knowledge Virtual Machines (ZKVMs) in general, which have widespread applications, particularly in blockchain networks.
- Despite the complexity of the task, which demands extensive knowledge in mathematics, RAM programs, and applied cryptography, particularly zero-knowledge proofs, Cao-Thang has diligently studied and acquired all the necessary background knowledge required for this project. He has thoroughly presented his findings, with a focus on zero-knowledge proofs and their variations.
- In this project, Cao-Thang managed to propose a generalization of memory for RAM programs, which provides flexibility as the users not only could configure the memory components to their desire but also easily develop custom instruction sets for their projects.
- In addition, based on previous works, Thang also formally defined the constraints for checking the consistency of the generalized memory above. Finally, Thang proposed to use Halo2, a ZKP protocol proposed by Zcash and developed by the PSE team of the Ethereum Foundation, to build a system that supports proving the consistency of the generalized memory.
- The system is analyzed carefully and experimented successfully for testing the run time and correctness of the main features.

6. Những thiếu sót chính của LV/ĐATN: There is no clear specification as to why the author decided to use Halo2 instead of other ZKP protocols because there are other ZKP protocols such as STARK, Plonky2, which could be great candidates as well. A table of comparison of these protocols leading to the decision to use Halo2, for example, would be great.
7. Đề nghị: [✓ Được bảo vệ] Bổ sung thêm để bảo vệ □ Không được bảo vệ □
8. Các câu hỏi SV phải trả lời trước Hội đồng:

a. There are various ZKP protocols out there that can be used to tackle the problem such as Groth16, STARK, Spartan, Plonk, and Plonky2. Why do you choose Halo2?
b. So far, some benefits of your generalized memory are presented, but there is no concrete example of it. Therefore, can you provide a concrete example of when your generalized memory proposal would be more beneficial than the traditional memory?

9. Đánh giá chung (bằng chữ: Xuất sắc, Giỏi, Khá, TB): **Xuất sắc** Điểm: **10** /10

Ký tên (ghi rõ họ tên)

[illegible]

Nguyễn An Khương

TRƯỜNG ĐẠI HỌC BÁCH KHOA	**CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM**
KHOA KH & KT MÁY TÍNH	**Độc lập - Tự do - Hạnh phúc**

Ngày 31 tháng 05 năm 2024

PHIẾU ĐÁNH GIÁ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP

(Dành cho người ~~hướng dẫn~~/phản biện)

1. Sinh viên thực hiện:
Trịnh Cao Thắng MSSV: 2014550 Ngành (chuyên ngành): Kỹ thuật Máy tính

2. Đề tài: Nghiên cứu và phát triển module quản lý và chứng thực bộ nhớ dựa trên kĩ thuật chứng minh không để lộ tri thức / Research and develop a universal memory commitment module based on zero-knowledge proofs

3. Họ tên người ~~hướng dẫn~~/phản biện: Huỳnh Hoàng Kha
4. Tổng quát về bản thuyết minh:

Số trang:	Số chương:
Số bảng số liệu	Số hình vẽ:
Số tài liệu tham khảo:	Phần mềm tính toán:
Hiện vật (sản phẩm)	

5. Những ưu điểm chính của LV/ ĐATN:

Sinh viên có hiểu biết về lĩnh vực Zero-Knowledge Proof và kỹ năng lập trình tốt, từ đó đã thành công phát triển một thư viện lập trình cho ngôn ngữ RUST. Việc tổ chức lại bộ nhớ chương trình bằng kiểu dữ liệu người dùng tự định nghĩa và thực hiện giám sát các tác vụ bộ nhớ nhằm phát hiện sự không nhất quán về dữ liệu có thể khiến cho tin tặc gặp nhiều khó khăn và tốn nhiều thời gian hơn khi thực hiện tấn công bằng phương pháp can thiệp bộ nhớ.

6. Những thiếu sót chính của LV/ĐATN:
- Thư viện được phát triển và sử dụng ở user's space, do đó chưa triệt để bảo vệ được bộ nhớ khỏi bị thay đổi bởi tin tặc.
- Chi phí phát sinh khi sử dụng thư viện chưa được đo đạc và báo cáo đầy đủ.
- Sinh viên cần cải thiện kỹ năng trình bày.

7. Đề nghị: Được bảo vệ ☒ Bổ sung thêm để bảo vệ □ Không được bảo vệ □

8. Các câu hỏi SV phải trả lời trước Hội đồng:
a. Sinh viên cho biết ý nghĩa và ưu thế của việc tổ chức lại dữ liệu chương trình bằng kiểu dữ liệu người dùng tự định nghĩa như đã đề xuất trong đồ án.
b. Có cơ chế nào để ngăn chặn hoặc phát hiện việc tin tặc tấn công thay đổi các giá trị trong khu vực bộ nhớ của memory operations log (nhật ký tác vụ bộ nhớ) hay không?
c. Các chi phí phát sinh (overhead) của việc lưu vết tác vụ bộ nhớ (memory operations logging) đã được đo lường và đánh giá hay chưa?

9. Đánh giá chung (bằng chữ: Xuất sắc, Giỏi, Khá, TB): Điểm : 8.0 /10

Ký tên (ghi rõ họ tên)

[illegible]

Huỳnh Hoàng Kha

Declaration

We certify that this capstone project is our research under the supervision of Dr. Nguyen An Khuong, Mr. Tran Anh Dung, and Mr. Pham Nhat Minh, which is derived from practical needs and our desire to study. We also declare that everything written in this report results from our research. Where I have talked about the work of others, this is always clearly stated. All statements taken literally from other writings or referred to by analogy are marked, and the source is always given.

Acknowledgement

I want to express my deepest gratitude to my supervisor, Dr. Nguyen An Khuong, for his invaluable guidance throughout this project. His mentorship introduced me to the fascinating field of cryptography, providing me with excellent resources to strengthen my understanding of mathematics and cryptography, especially the foundational knowledge essential for this project. Dr. Khuong's guidance was instrumental in navigating the challenges I encountered across nearly all aspects of my thesis. Beyond academics, he offered me significant advice and support during difficult times in my personal life.

I want to extend my gratitude to Mr. Tran Anh Dung and Mr. Pham Nhat Minh from Orochi Network, who proposed this topic and technically guided me to the very end. This project was conducted during the student's internship at Orochi Network from 12 June 2023 to 11 June 2024. This thesis is a part of a big project which is a property of Orochi Network. I also want to thank Mr. Tang Khai Hanh, MSc, for the valuable exchanges on this topic. I am sincerely grateful for the opportunities to learn and grow. Their willingness to share their expertise and provide constructive feedback has been pivotal in shaping my professional development and completing this capstone project. The challenging projects, the collaborative work environment of Orochi Network, and the chance to contribute to real-world initiatives have genuinely deepened my understanding of the career.

My sincere thank is also extended to all of my friends I have met and kept good contact including Nguyen Trong Tin, Vo Dang Thi, and Nguyen Khoa Nam, especially Tran Gia Tuan, Tran Pham Luong, Le Nhut Huy, Nguyen Van Son, and Nguyen Hoang Huy, who helped me a lot in studying many subjects at university and listened to me when I had so much stress, especially during my last three years. With them, I was able to succeed many subjects and possibly this capstone project.

Last but not least, I would like to thank my parents for their unconditional love and support during my career.

Abstract

The main aim of this specialised project is to build a memory management module that executes basic RAM programs and to prove that the execution of the RAM program is correct and the memory is consistent during the execution in zero-knowledge (i.e. proving the execution of the RAM program is correct without disclosing any relevant information about the memory itself).

Our proposal consists of the following steps: First, we propose to use an available generalisation of the memory by specifying the memory layout, memory components, and configurable memory word size. Then, we introduce basic memory instructions and specify an execution trace capturing the whole memory access process. We then propose to use two commitment schemes to manage the execution trace for opening to a verifier later. We also propose a high-level constraint system to prove the consistency of the memory in any RAM program execution. Finally, we demonstrate an available implementation using the Halo2 proving system written in Rust programming language.

Some backgrounds in relevant mathematics, cryptography, and RAM programs are presented in this project.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Zero-knowledge proofs are one of the most crucial building blocks for many privacy-preserving systems. The basic concept of a zero-knowledge proof is to prove that a particular statement is true without revealing any information other than the fact that the statement is true. Goldwasser, Micali, and Rackoff first introduced zero-knowledge proof [GMR85], and in recent years, zero-knowledge proofs have had a remarkable impact on cryptography and Blockchain systems.

Over recent years, intense research has been done to construct different zero-knowledge protocols, each with varying tradeoffs between some efficiency measures. In blockchain applications, the focus is on *succinct proofs* of small statements, researched in [GGPR13, Gro16, Set20]. A succinct proof is a proof that allows a negligible probability that the proof incorrectly passes the verification (i.e. the proof is accepted as good although the computation was done incorrectly) in exchange for an easy and fast communication time between the prover and the verifier, often sublinearly scales with the size of the prover's witness [EA23]. Other research focused on zero-knowledge proof systems that improve the performance of the prover [JKO13, AHIV17]. Some research constructed zero-knowledge proof systems that can prove massive statements efficiently, such as Wolverine [WYKW21] and Quicksilver [YSWW21]. However, those early works mainly focused on proving statements that are represented as a *circuit (Boolean circuits or arithmetic circuits)*, which made those systems harder to apply when significantly large statements are intended to run in a *random-access machine (RAM)* computation model. Note that the RAM computation model is different from the RAM architecture of a general computer; hence, it is not related to any security issues. Therefore, newer research works began to construct zero-knowledge proof systems in the RAM model of computation, such as proving that the memory during the execution of a program in the RAM model (also called a *RAM program*) is consistent and proving that the RAM program is correctly executed. However, their constructions introduced limitations such as performance issues or the requirement of a trusted setup. Nonetheless, those research works ultimately remain pioneering, and the topic continues to hold significant potential for the future. Therefore, we decided to study zero-knowledge proofs in the RAM computation model and attempt to reduce the limitations that prior work introduced.

Overall, we aim to present a solution to prove the consistency of the memory of a RAM program in zero knowledge. Specifically, we aim to have a detailed generalisation of the memory in a RAM program (the components and the layout of the memory, how basic instructions interact with the memory, and how we can form an execution trace that records all memory accesses during execution) and propose to use some commitment schemes to commit useful data for the solution to proving the memory consistency and the correct execution of the RAM program. Ultimately, we will demonstrate an available implementation of the whole solution as a module that executes basic instructions in the RAM computation model and proves the memory consistency of the RAM program, which achieves completeness, soundness, and zero knowledge.

We aim to build the memory management module using a friendly programming language. It can serve as a utility module to any general zero-knowledge virtual machines (zkVMs), and can ultimately support zkVMs written in any programming language. It can also be used as a backend for a file system or a virtual machine database, which is compiled to run the virtual machine. There are currently many projects and organisations that focus on researching and implementing zkVMs, including the Privacy and Scaling Exploration[1], Risc Zero[2], Cairo [GPR21], and Orochi Network[3]. We will explain both zero-knowledge virtual machines and quickly review the available implementations in the Related works in Chapter 3.

[1]

[2]https://dev.risczero.com/api/zkvm/

[3]https://www.orochi.network/

1.2 Objectives and Scopes

In this capstone project, we aim to propose a solution for implementing a universal memory management module as follows:

- Propose a generalisation of the memory of a RAM program with memory storage, stack, and registers, along with basic memory instructions of a RAM program such as READ, WRITE, PUSH, and POP and an execution trace that records all memory accesses during the execution.

- Propose to use two commitment schemes to commit and create proofs for the execution trace, namely, Merkle Proof-of-inclusion for Merkle Tree and KZG polynomial commitment scheme. In this project, we intend to implement the KZG polynomial commitment scheme using the utilities of the Halo2 proving system implemented by the Privacy-Scalable Exploration team (PSE).

- Propose a constraint system for handling the memory consistency of a RAM program based on the work of Franzese et al [FKL+21]. In this project, we implement the translation from the proposed constraint systems into the circuit to check the memory's consistency using the Halo2 proving system.

With these requirements in mind, the scope of this thesis would include the following objectives:

- Derive a memory specification in the original definition of a RAM program from [FKL+21] and [dSGOTV22]. Note that the project does not consider some complex aspects of memory management, such as memory paging, concurrency, memory sharing, deadlock, etc.

- Understand and apply the Halo2 proving system to our implementation.

- Study commitment schemes, especially KZG commitment scheme [KZG10] to apply to our proposed commitment scheme.

1.3 Structures

The structure of this capstone project consists of 8 chapters, which are described as follows:

- **Chapter 1** gives an introduction and the objectives of this capstone project.
- **Chapter 2** summarises the required background materials about algebra, cryptography, and RAM programs, which are needed in this project.
- **Chapter 3** gives an overview of some initial and recent works on RAM programs and lists their advantages and disadvantages.
- **Chapter 4** overviews the Halo2 proving system we use to prove memory consistency and implement the solution.
- **Chapter 5** proposes and analyses a solution to build a memory management module based on zero-knowledge proofs.
- **Chapter 6** proposes a constraint system to prove the memory consistency problem.
- **Chapter 7** explains the implementation of component modules in an available solution using Rust and Halo2 proving system called the project zkMemory[4] from Orochi Network[5].
- **Chapter 8** provides some results of unit tests for the whole system as well as the prover and verifier time performance of proving the memory consistency of a RAM program.
- **Chapter 9** gives conclusions of the capstone project and future directions in the future.

[4]https://github.com/orochi-network/orochimaru/tree/main/zkmemory
[5]https://orochi.network/

Chapter 2

Background

This chapter explains various background materials in algebra, mathematical cryptography, and RAM programs. These materials are essential for analyzing the existing works on RAM programs that are described in Chapter 3 and proposing a solution to the problem in Chapter 5.

2.1 Algebra

2.1.1 Notations

Vectors. We denote vectors by bolded lowercase letters such as $\mathbf{u}, \mathbf{v}$. We often use $(u_1, u_2, \cdots, u_n)$ to denote a vector whose components are $u_1, u_2, \cdots, u_n$. We denote the i-th component of a vector u by u_i, while we often use $u[i]$ in algorithms. We denote the inner product (or the dot product) of 2 vectors $\mathbf{u}$ and $\mathbf{v}$ by $\mathbf{u} \cdot \mathbf{v}$.

Sets. We use letters such as $\mathcal{A}, \mathcal{B}, \mathcal{C}$ to denote a set. Sometimes we also denote a set by $\{a_1, a_2, \cdots, a_n\}$ or $\{a_i\}_{i=1}^{n}$.

Algebra. We denote by $f(x)$ a polynomial in x and f_i to be the coefficients of $f(x)$ (note that f_0 is the coefficient of the smallest degree, also called the *constant term* of the polynomial). We denote a group by $\mathbb{G}$ and denote by p the order of its group if $\mathbb{G}$ is a finite group.

Algorithms. We use $Y \leftarrow \mathsf{Alg}(X)$ to denote the algorithm that takes X as input and outputs Y. We also use $Y \overset{\$}{\leftarrow} \mathcal{S}$ to denote that Y is randomly chosen from the set $\mathcal{S}$.

Arithmetic circuits/sub-circuits. We denote the arithmetic circuit by C_f where f is the functionality of the arithmetic circuit. We denote arithmetic sub-circuits in the same way as denoting algorithms such as $\mathsf{SubCircuit}([x])$. We denote by $[x]$ wire values in an arithmetic circuit or sub-circuit that should remain hidden when the circuit is proven in zero-knowledge.

Protocols. We denote by $P_1, P_2, \cdots, P_n$ the participants in a protocol where n is the number of participants. We denote by $\mathsf{Protocol}(x)\langle A(w_A), B(w_B)\rangle$ the interactive protocol named $\mathsf{Protocol}$ between two participants A and B with a common input x where A holds a secret value w_A, and B holds a secret value w_B.

Cryptography. We denote λ to be the security parameter. We denote sk, pk as the secret and public keys, respectively. We denote H_λ to be the cryptographic hash function used in certain cryptographic systems. We denote the concatenation operator by $\|$. For example, $H_\lambda(A \parallel B)$ denotes the hash output of the concatenation between A and B, where A and B are in byte representations. Finally, we denote $\mathcal{A}$ to be an adversary who attempts to break the security of a cryptosystem or a cryptographic assumption.

2.1.2 Group Theory

Definition 2.1.1 (Groups)**.** A **group** is a set $\mathbb{G}$ together with a binary operation $\cdot$ on $\mathbb{G}$ that satisfies all of the following properties:

- **Closure.** For all $a, b \in \mathbb{G}$, we have: $a \cdot b \in \mathbb{G}$.
- **Associativity.** For all $a, b, c \in \mathbb{G}$, we have: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
- **Identity.** There exists an element $e \in \mathbb{G}$ (called the **identity element**) such that for all $a \in \mathbb{G}$, we have: $a \cdot e = e \cdot a = e$.
- **Inverse.** For all $a \in \mathbb{G}$, there exists $a' \in \mathbb{G}$ (called the **inverse of** a) such that: $a \cdot a' = a' \cdot a = e$.

Example 2.1.1. *Consider the set of rational numbers $\mathbb{Q} = \{a/b : a, b, \in \mathbb{Z}, b \neq 0\}$ under addition. The set $\mathbb{Q}$ forms a group, with 0 being the identity, and $-a/b$ being the inverse of a/b.*

Definition 2.1.2 (Abelian groups)**.** A group $\mathbb{G}$ is said to be an **abelian group** if $\mathbb{G}$ also satisfies the property of **commutativity**: For all $a, b \in \mathbb{G}$, we have: $a \cdot b = b \cdot a$.

Example 2.1.2. *For each integer n, the set $n\mathbb{Z} = \{nz : z \in \mathbb{Z}\}$ forms an abelian group, with 0 being the identity, and $n(-z)$ being the inverse of nz.*

Definition 2.1.3 (Cyclic groups)**.** Let $(\mathbb{G}, \cdot)$ be a group, then G is called a **cyclic group** if there exists an element $g \in \mathbb{G}$ such that every element $a \in \mathbb{G}$ can be expressed as $g^k = \underbrace{g \cdot g \cdot \cdots \cdot g}_{k\text{-times}}$ for some integer k. We call such an element a **generator** of the group $\mathbb{G}$.

Example 2.1.3. *Consider the set $\mathbb{Z}_7^*$ of positive integers modulo 7 under multiplication. We can verify that $3^1 = 3, 3^2 = 2, 3^3 = 6, 3^4 = 4, 3^5 = 5, 3^6 = 1$. Therefore, the set $\mathbb{Z}_7^*$ under multiplication forms a cyclic group with 3 being its generator.*

Definition 2.1.4 (Homomorphisms)**.** A **homomorphism** is a function f from a group $(\mathbb{G}, \cdot)$ to a group $(\mathbb{G}', *)$ such that:

$$f(a \cdot b) = f(a) * f(b) \text{ for all } a, b \in \mathbb{G}.$$

In particular, the function f maps the identity $e_{\mathbb{G}}$ of $\mathbb{G}$ to the identity $e'_{\mathbb{G}}$ of $\mathbb{G}'$, thus we have $f(e_{\mathbb{G}}) = e'_{\mathbb{G}}$.

Example 2.1.4. *Consider the set $\mathbb{Z}$ under addition, the set $\mathbb{Z}_7^*$ defined in Example 2.1.3, and the function $f : \mathbb{Z} \longrightarrow \mathbb{Z}_7^*$ given by $f(x) = 3^x \mod 7$. Then f is a homomorphism. In fact, for all $a, b \in \mathbb{Z}$, we have*

$$f(a) \cdot f(b) = 3^a \cdot 3^b \pmod 7 = 3^{a+b} \pmod 7 = f(a + b).$$

Example 2.1.5. *Consider the set $\mathbb{Z}_8$ under addition , the set $\mathbb{Z}_7^*$ under multiplication, and the function $g : \mathbb{Z}_8 \longrightarrow \mathbb{Z}_7^*$ given by $f(x) = 3^x \mod 7$. Then f is NOT a homomorphism as we can find a counterexample as follows:*

$$g(0) = g(2 + 6) \pmod 8 = g(2) \cdot g(6) = 3^2 \cdot 3^6 = 3^8 \pmod 7 = 2 \neq 1.$$

Definition 2.1.5 (Isomorphisms)**.** A homomorphism $f : \mathbb{G} \to \mathbb{G}'$ is an **isomorphism** if f is a bijection from $\mathbb{G}$ to $\mathbb{G}'$. If such an isomorphism f exists, then $\mathbb{G}$ is **isomorphic** to $\mathbb{G}'$, denoted by $\mathbb{G} \cong \mathbb{G}'$.

Example 2.1.6. *Consider the set $\mathbb{Z}_4$ under addition, the set $\mathbb{Z}_5^*$ under multiplication, and the function $f : \mathbb{Z} \longrightarrow \mathbb{Z}_7^*$ given by $f(x) = 3^x \mod 5$. We have $f(0) = 1, f(1) = 3, f(2) = 4, f(3) = 2$. Therefore, f is a bijection from $\mathbb{Z}_4$ to $\mathbb{Z}_5^*$. Also, f is a homomorphism. In fact,* *for all $a, b \in \mathbb{Z}_4$, we have*

$$f(a) \cdot f(b) = 3^a \cdot 3^b \pmod 5 = 3^{a+b} \pmod 5 = f(a+b).$$

Therefore, f is an isomorphism and we write $\mathbb{Z}_4 \cong \mathbb{Z}_5^*$.

2.1.3 Field Theory

Definition 2.1.6 (Fields)**.** A set $\mathbb{F}$ together with addition operation "+" and multiplication operation "·" is called a **field** if it satisfies these properties:

- **Abelian group under additions.** The set $\mathbb{F}$ under addition forms an abelian group with an additive identity of 0.
- **Abelian group under multiplications.** The set $\mathbb{F} \backslash \{0\}$ under multiplication forms an abelian group with a multiplicative identity of $1 \neq 0$.
- **Distributivity.** We have $a \cdot (b + c) = a \cdot b + a \cdot c$ for all $a, b, c, \in \mathbb{F}$.

Example 2.1.7. *We consider some common sets and determine whether each set is a field.*

(a) Consider the set $\mathbb{Q}$, with addition and multiplication defined as usual. $\mathbb{Q}$ under addition forms an abelian group with identity 0. The set $\mathbb{Q}$ under multiplication also forms an abelian group with identity 1. The set $\mathbb{Q}$ also satisfies the distributive property. Hence, $\mathbb{Q}$ is a field.

(b) Consider the set $\mathbb{Z}$, with addition and multiplication defined as usual. The set $\mathbb{Z}$ under addition forms an abelian group with identity 0. However, only $1, -1$ have their multiplicative inverses, which are $1, -1$ respectively. Therefore, $\mathbb{Z}$ is not a field.

(c) The set $\mathbb{F}_p = \{0, 1, \dots, p-1\}$ is a field, and it has finite elements, p elements to be precise. Thus, $\mathbb{F}_p$ is called a ***finite field****.*

Definition 2.1.7 (Subfields [Sho05])**.** Given a field $\mathbb{F}$ and a subset $\mathbb{F}' \subset \mathbb{F}$. If $\mathbb{F}'$ is a field under addition and multiplication defined from $\mathbb{F}'$, then $\mathbb{F}'$ is called a **subfield** of $\mathbb{F}$, or $\mathbb{F}$ is an **extension field** of $\mathbb{F}'$.

Example 2.1.8. *$\mathbb{Q}$ is a subfield of $\mathbb{R}$ and $\mathbb{R}$ is a subfield of the field of complex numbers $\mathbb{C}$.*

2.1.4 Bilinear Pairings

In this section, we define the pairing operation which is used in constructions of polynomial commitment schemes in Section 2.3.6.2 by following the definition from [KZG10].

Definition 2.1.8 (Bilinear pairings [KZG10])**.** For three cyclic groups $\mathbb{G}$, $\hat{\mathbb{G}}$, and $\mathbb{G}_T$ (all of which we shall write multiplicatively) of the same prime order p, a **bilinear pairing** e is a map $e : \mathbb{G} \times \hat{\mathbb{G}} \to \mathbb{G}_T$ with the following properties:

- **Bilinearity**: for $g \in \mathbb{G}$ and $\hat{g} \in \hat{\mathbb{G}}$ and $a, b \in \mathbb{Z}_p$, we have

$$e(g^a, \hat{g}^b) = e(g, \hat{g})^{ab}.$$

- **Non-degeneracy**: The map does not send all pairs $\mathbb{G} \times \hat{\mathbb{G}}$ to unity $e_T \in \mathbb{G}_T$. In particular

$$e(g, \hat{g}) = e_T \text{ for all } \hat{g} \in \hat{\mathbb{G}} \Rightarrow g = e.$$

If there exists an efficient algorithm to compute $e(g, \hat{g})$ for any $g \in \mathbb{G}$ and $\hat{g} \in \hat{\mathbb{G}}$, then the pairing e is called **admissible**. All pairings that we consider are **admissible** and **not feasible to invert**. We call such groups $\mathbb{G}$ and $\hat{\mathbb{G}}$ **pairing-friendly groups**.

Example 2.1.9. *Consider the three groups $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}_T = (\mathbb{Z}, +)$ and a map $e : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T$ given by $e(x, y) = x \cdot y$. We check the two properties of the map e to determine if e is a bilinear pairing as follows:*

***Bilinearity**: For all $a, b \in \mathbb{Z}$, it holds that*

$$e(x \cdot a, b \cdot y) = x \cdot a \cdot y \cdot b = e(x, y) \cdot (ab).$$

***Non-degenerate**: $e(x, y) = 1$ if and only if $x = y = 1$ or $x = y = -1$.*

There are 3 types of pairings: namely, type 1, type 2 and type 3, described as follows:

- Type 1 : An isomorphism $\phi : \hat{\mathbb{G}} \to \mathbb{G}$ as well as its inverse ϕ^{-1} are efficiently computable. These are also called **symmetric pairings** as for such pairings $e(g, \hat{g}) = e(\phi(\hat{g}), \phi^{-1}(g))$ for any $g \in \mathbb{G}$ and $\hat{g} \in \hat{\mathbb{G}}$.

- Type 2 : Only the isomorphism ϕ, but not ϕ^{-1} is efficiently computable.

- Type 3 : Neither ϕ nor ϕ^{-1} can be efficiently computed.

2.1.5 Lagrange Interpolation Polynomial

In this section, we define the basic concepts of Lagrange interpolation polynomial, which will be used in our solution for KZG commitment schemes in Section 5. The definition of commitment schemes is also explained in Section 2.3.6. Note that in this section, we define coordinates of a point (x, y) as **node** and **value** respectively.

Definition 2.1.9 (Lagrange basis)**.** Given a set of $k+1$ nodes $\{x_0, x_1, ..., x_k\}$ with $x_i \neq x_j$ for all $i \neq j$, the **Lagrange basis** for the polynomial of degree $\leq k$ for these nodes is the set $L_B = \{\ell_0(x), \ell_1(x), ..., \ell_k(x)\}$, each of degree k such that $\ell_i(x_j) = 0$ if $i \neq j$ and $\ell_i(x_i) = 1$. Each **Lagrange basis polynomial** of L_B can be expressed by the product

$$\ell_i(x) = \prod_{\substack{0 \leq m \leq k \\ m \neq i}} \frac{x - x_m}{x_i - x_m} = \frac{(x - x_0)}{(x_i - x_0)} \cdots \frac{(x - x_{j-1})}{(x_i - x_{j-1})} \frac{(x - x_{j+1})}{(x_i - x_{j+1})} \cdots \frac{(x - x_k)}{(x_i - x_k)}.$$

Definition 2.1.10 (Lagrange Interpolation Polynomials)**.** Given a set of $k+1$ nodes $\{x_0, x_1, ..., x_k\}$ with $x_i \neq x_j$ for all $i \neq j$ and their corresponding values $\{y_0, y_1, ..., y_k\}$, the **Lagrange interpolation polynomial** for those nodes and their values is a polynomial $L(x)$ of degree $\leq k$ such that $L(x_i) = y_i$ for all $0 \leq i \leq k$. The Lagrange interpolation polynomial is computed as the linear combination

$$L(x) = \sum_{i=0}^{k} y_i \ell_i(x).$$

Example 2.1.10. *Consider the field $\mathbb{Q}$ and given 4 points $(1, -18), (3, -68), (7, -72), (9, 70)$. We will find a polynomial of degree ≤ 3 that interpolates all 4 said points.*
We compute the Lagrange basis L_B as follows:

- $\ell_0(x) = \prod_{\substack{0 \leq m \leq 3 \\ m \neq 0}} \frac{x - x_m}{x_0 - x_m} = \frac{x-3}{1-3} . \frac{x-7}{1-7} . \frac{x-9}{1-9} = \frac{-1}{96}x^3 + \frac{19}{96}x^2 - \frac{37}{32}x + \frac{63}{32}.$

- $\ell_1(x) = \prod_{\substack{0 \leq m \leq 3 \\ m \neq 0}} \frac{x - x_m}{x_1 - x_m} = \frac{x-1}{3-1} . \frac{x-7}{3-7} . \frac{x-9}{3-9} = \frac{1}{48}x^3 - \frac{17}{48}x^2 + \frac{79}{48}x - \frac{21}{16}.$

- $\ell_2(x) = \prod_{\substack{0 \leq m \leq 3 \\ m \neq 0}} \frac{x - x_m}{x_2 - x_m} = \frac{x-1}{7-1} . \frac{x-3}{7-3} . \frac{x-9}{7-9} = \frac{-1}{48}x^3 + \frac{13}{48}x^2 - \frac{13}{16}x + \frac{9}{16}.$

- $\ell_3(x) = \prod_{\substack{0\le m\le 3\\ m\neq 0}} \frac{x-x_m}{x_3-x_m} = \frac{x-1}{9-1}.\frac{x-3}{9-3}.\frac{x-7}{9-7} = \frac{1}{96}x^3 - \frac{11}{96}x^2 + \frac{31}{96}x - \frac{7}{32}.$

Then, we compute the Lagrange interpolation polynomial

$$L(x) = \sum_{i=0}^{3} y_i\ell_i(x) = x^3 - 7x^2 - 10x - 2.$$

Finally, We check that the result polynomial is correct by evaluating the polynomial at $1, 3, 7$ *and* 9*:* $L(1) = -18, L(3) = -68, L(7) = -72, L(9) = 70$.

2.2 Elliptic Curves

In this section, we explain some fundamental concepts of elliptic curves defined on finite fields. Most of the materials in this section are referenced from [HPS08].

2.2.1 Definitions

Definition 2.2.1 (Elliptic curves)**.** Let $\mathbb{F}_p$ be a finite field where p is an odd prime. We define an **elliptic curve** over the finite field $\mathbb{F}_p$ to be an equation of the form

$$E : X^2 = Y^3 + AX + B \qquad \text{with } A, B \in \mathbb{F}_p \text{ satisfying } 4A^3 + 27B^2 \neq 0,$$

and consider all the points on E with coordinates in $\mathbb{F}_p$, denoted by

$$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p^2 : y^2 = x^3 + Ax + B\} \cup \{\mathcal{O}\}$$

where $\mathcal{O}$ is called the **point at infinity**.

Elliptic curves over the finite field $\mathbb{F}_2$ are more complicated to explain and thus they are beyond this scope. Therefore, in the definition of elliptic curves, we assume that $p > 3$.

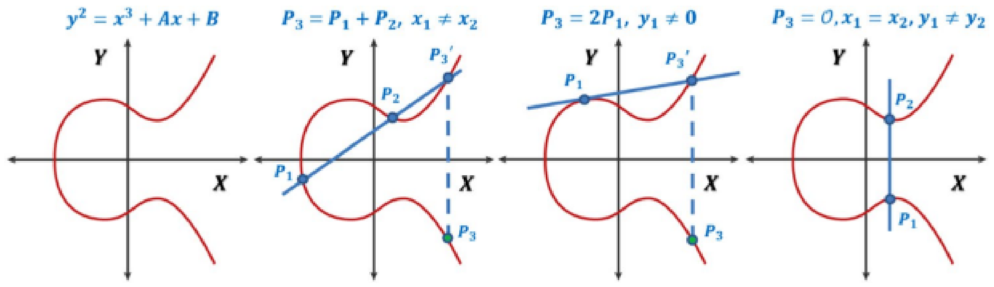
We define the **addition law** on elliptic curve E. Let $P(x_1, y_1), Q(x_2, y_2) \in E$, we can add P and Q to get their sum as follows:

1. Let L be the line connecting P and Q, or the tangent line of E at P if $P = Q$.

2. Let R be the intersection of L and the curve E. If L and E do not have an intersection other than P and Q, then $R = \mathcal{O}$.

3. Reflect the point R across the X-axis (which is also called **point negation**) to get R'. Then R' is the sum of P and Q. If $R = \mathcal{O}$ then $R' = \mathcal{O}$.

The addition law on elliptic curves can be illustrated by Figure 2.1:



$y^2 = x^3 + Ax + B$
Y
X
$P_3 = P_1 + P_2,\ x_1 \neq x_2$
Y
P_3'
P_2
P_1
X
P_3
$P_3 = 2P_1,\ y_1 \neq 0$
Y
P_3'
P_1
X
P_3
$P_3 = \mathcal{O}, x_1 = x_2, y_1 \neq y_2$
Y
P_2
X
P_1

Figure 2.1: *An intuitive representation of the addition law on an elliptic curve [AAEHR22]*

2.2.2 Formulas for The Addition Law

We now express the general formula of the addition law in elliptic curves. Let E be an elliptic curve and $P(x_1, y_1), Q(x_2, y_2)$ are two points in the curve E. We consider two cases: $P \neq Q$ and $P = Q$.

Let $L : y = \lambda x + \beta$ be the line that intersects P and Q. Then the slope λ of the line L is calculated as

$$\lambda = \begin{cases} \dfrac{y_1 - y_2}{x_1 - x_2} & \text{if } P \neq Q \\ \dfrac{3x_1^2 + a}{2y_1} & \text{if } P = Q. \end{cases}$$

We consider the intersection between the line L and the curve E. We get this equation

$$x^3 - \lambda^2 x^2 - 2\lambda x\beta - \beta^2 + ax + b = 0.$$

Since the equation should have 3 solutions x_1, x_2, x_3 including P, Q, we have

$$x_1 + x_2 + x_3 = \lambda^2,$$

so the point $R = L \cap E$ has coordinates

$$x_3 = \begin{cases} \lambda^2 - x_1 - x_2 & \text{if } P \neq Q \\ \lambda^2 - 2x_1 & \text{if } P = Q \end{cases}$$
$$y_3 = \lambda(x_3 - x_1) + y_1.$$

There, we reflect the point R across the X-axis, which outputs the point $R' = -R = (x_3, -y_3)$. This point R' is the sum of two points P and Q.

Example 2.2.1. *Consider the curve $E : y^2 = x^3 + 7x + 4$ over the finite field $\mathbb{F}_{31}$ and two points $P = (28, 24), Q = (18, 17) \in E$. We will find the sum $P + Q$ and $P + P = 2.P$.*

- *The point $R = P + Q$ is calculated using the formula as follows:*
  - $\lambda \equiv \frac{y_1 - y_2}{x_1 - x_2} \equiv \frac{24 - 17}{28 - 18} \equiv 10 \pmod{31}$.
  - $x_3 \equiv \lambda^2 - x_1 - x_2 \equiv 10^2 - 28 - 18 \equiv 23 \pmod{31}$.
  - $y_3 \equiv \lambda(x_3 - x_1) + y_1 \equiv 10.(23 - 28) + 24 \equiv 5 \pmod{31}$.
  - *The point $R(x_3, -y_3) = (23, 26)$ is the sum.*
- *The point $S = P + P = 2.P$ is calculated using the formulas as follows:*
  - $\lambda \equiv \frac{3x_1^2 + a}{2y_1} \equiv \frac{3.28^2 + 7}{2.24} \equiv 2 \pmod{31}$.
  - $x_3 \equiv \lambda^2 - 2x_1 \equiv 2^2 - 2.28 \equiv 10 \pmod{31}$.
  - $y_3 \equiv \lambda(x_3 - x_1) + y_1 \equiv 2.(10 - 28) + 24 \equiv 19 \pmod{31}$.
  - *The point $R(x_3, -y_3) = (10, 12)$ is the sum.*

We now explain the properties of the addition law as follows:

Definition 2.2.2 (Addition law properties)**.** Let E be an elliptic curve. Then the addition law on E satisfies these properties:

(a) **Identity**. $P + \mathcal{O} = \mathcal{O} + P = P$ for all $P \in E$.

(b) **Inverse**. $P + (-P) = \mathcal{O}$ for all $P \in E$.

(a) **Associativity** $(P + Q) + R = P + (Q + R)$ for all $P, Q, R \in E$.

(a) **Commutativity** $P + Q = Q + P$ for all $P, Q \in E$.

In other words, the addition law makes $(E(\mathbb{F}_p), +)$ an abelian group.

2.2.3 Discrete Logarithm Problems based on Elliptic Curves

We now explain the elliptic curve discrete logarithm problem. We introduce an operation of elliptic curve E, which is called **scalar multiplication**, as follows:

$$n.P = \underbrace{P + P + \cdots\cdots + P}_{n\text{-times}}.$$

If n is negative, then $n.P = -n.(-P)$.

The **discrete logarithm** problem is considered to be hard on finite fields and also on the group of an elliptic curve. The problem is defined as follows:

Definition 2.2.3 (Elliptic Curve Discrete Logarithm Problem (ECDLP) [HPS08])**.** Let E be an elliptic curve over the finite field $\mathbb{F}_p$ and let P and Q be points in $E(\mathbb{F}_p)$. The **Elliptic Curve Discrete Logarithm Problem (ECDLP)** is the problem of finding the integer n such that $Q = nP$. The integer n is called *the elliptic discrete logarithm of* Q *with respect to* P.

Example 2.2.2. *Consider an elliptic curve* $E : y^2 = x^3 + 3x + 13$ *over the finite field* $\mathbb{F}_7$ *and two points* $P(3, 1) \in E$ *and* $Q = nP = (2, 3) \in E$*. We can find the integer* n *by listing all the points* $\{iP : i = 1..8\}$ *until we find a collision with the point* Q*. We find that* $nP = Q = (2, 3)$ *with* $n = 4$*.*

The elliptic curve discrete logarithm problem is considered hard on the group of elliptic curves. The fastest known algorithm to solve ECDLP in $E(\mathbb{F}_p)$ takes approximately $\sqrt{p}$ steps [HPS08]. Therefore, elliptic curves are widely used for cryptographic constructions because they provide a comparable security level with a smaller key size required compared to the RSA cryptosystem. Table 2.1 presents an overview of security levels between Elliptic curves and RSA (which is based on the problem of integer factorisation).

Table 2.1: Security bit-level comparison between RSA and ECC [Bar16].

Security Bit Level	RSA	ECC
≤ 80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

2.3 Cryptography

This section explains some of the fundamental concepts in cryptography, which are all important materials for Chapter 3, Chapter 4, Chapter 5, and 6.

2.3.1 Negligible Functions

This term will be defined formally as the rest of the project, we often use the term "negligible functions" when defining security properties for many cryptographic primitives.

Definition 2.3.1 (Negligible functions)**.** A function $\mathsf{negl} : \mathbb{N} \longrightarrow \mathbb{R}^+$ is called **negligible** if for any polynomial $f(x)$, $\mathsf{negl}(x) < 1/f(x)$ for all sufficiently large $x \in \mathbb{R}^+$. Otherwise, negl is called **non-negligible**.

Example 2.3.1. *The function* $\mathsf{negl}(x) = \dfrac{1}{x^x}$ *is negligible. The function* $\mathsf{negl}(x) = \dfrac{1}{3x + x^5}$ *is non-negligible.*

2.3.2 Cryptographic Hardness Assumptions

In cryptography, numerous problems are considered to be hard to solve for a computationally bounded algorithm. These problems are the foundations for constructing other cryptosystems and are used in the security proofs for the corresponding cryptosystems. In this section, we focus on the two assumptions: the **Discrete logarithm (DL) Assumption** and the t**-Strong Diffie-Hellman (**t**-SDH) Assumption** in [KZG10]. These assumptions will be used as security proofs for the construction of a polynomial commitment scheme.

Let λ be a security parameter and p be a prime with $p = \Omega(2^\lambda)$. Given a cyclic group $\mathbb{G}$ with order p and a generator g. The two following assumptions can be defined as follows:

Discrete Logarithm (DL) Assumption. For every adversary $\mathcal{A}_{DL}$ with polynomial running time in λ, it holds that:

$$\Pr\left[\ \mathcal{A}_{DL}(g, g^a) = a \ \middle|\ a \xleftarrow{\$} \mathbb{Z}_p^*\ \right] = \mathsf{negl}(\lambda).$$

Informally, for uniformly chosen $a \in \mathbb{Z}_p^*$ and given g, g^a, it is computationally hard to find such an a, except for negligible probability.

t-Strong Diffie-Hellman (t-SDH) Assumption. Given as input a $(t+1)$-tuple $(g, g^{\alpha}, \ldots, g^{\alpha^t}) \in \mathbb{G}^{t+1}$, for every adversary $\mathcal{A}_{t\text{-SDH}}$ with polynomial running time in λ, it holds that:

$$\Pr\left[\ \mathcal{A}_{t\text{-SDH}}(g, g^{\alpha}, \ldots, g^{\alpha^t}) = (c, g^{\frac{1}{\alpha+c}})\ \middle|\ a \xleftarrow{\$} \mathbb{Z}_p^*\ \right] = \mathsf{negl}(\lambda).$$

Informally, for uniformly chosen $a \in \mathbb{Z}_p^*$ and given a $(t+1)$-tuple $(g, g^{\alpha}, \ldots, g^{\alpha^t}) \in \mathbb{G}^{t+1}$, it is computationally hard to compute $g^{\frac{1}{\alpha+c}}$ where c is some integer that the adversary chooses.

2.3.3 Cryptographic Hash Functions

A cryptographic hash function is a hash algorithm having many desirable properties for cryptographic applications. It is most commonly used in information security. In this section, we formally define a cryptographic hash function.

Definition 2.3.2 (Cryptographic hash function)**.** A **cryptographic hash function** $H_{\lambda} : \{0,1\}^* \longrightarrow \{0,1\}^{\lambda}$ takes as input an arbitrary length bit string and outputs a fixed length bit string such that it satisfies all the following properties:

1. **Efficient Evaluation.** There exists a polynomial running time algorithm in λ such that, given X, returns $H_{\lambda}(X)$.

2. **Pre-image Resistance.** Given a hash output $Y = H_{\lambda}(X)$, it is computationally infeasible to find X. More formally, there exists a negligible function negl such that

$$\Pr\left[\ Y = H_{\lambda}(X)\ \middle|\ X \leftarrow \mathcal{A}(\lambda, Y)\ \right] = \mathsf{negl}(\lambda).$$

3. **Second Pre-image Resistance.** Given only an input X, it is computationally infeasible to find another input X' such that $H_{\lambda}(X) = H_{\lambda}(X')$. More formally, there exists a negligible function negl such that

$$\Pr\left[\begin{array}{l} X' \neq X \wedge \\ H_{\lambda}(X') = H_{\lambda}(X) \end{array}\ \middle|\ X' \leftarrow \mathcal{A}(\lambda, X)\ \right] = \mathsf{negl}(\lambda).$$

4. **Collision Resistance.** It is computationally infeasible to find two different inputs X, X' such that $H_{\lambda}(X) = H_{\lambda}(X')$. More formally, there exists a negligible function negl

such that
$$\Pr\left[\begin{array}{l} X' \neq X \wedge \\ H_\lambda(X') = H_\lambda(X) \end{array} \middle| \ (X, X') \leftarrow \mathcal{A}(\lambda) \right] = \mathsf{negl}(\lambda).$$

2.3.4 Zero-knowledge Proofs

A **zero-knowledge proof** is a protocol involving two participants: the PROVER P and the VERIFIER V. Informally, a zero-knowledge proof allows the prover to convince the verifier that a certain statement is true without revealing further information other than the fact that the statement is true.

We now give an example that helps us understand the concept of ZKP intuitively by [HPS08].

Example 2.3.2. *Considering two persons jointly executing a zero-knowledge proof protocol as follows: Peggy the* PROVER *and Victor the* VERIFIER. *Peggy chooses two large primes p and q and publishes their product $N = pq$. Peggy's task is to prove to Victor that a certain number y is a square modulo N without revealing any other data that can allow Victor to prove to other people that y is a square modulo N. Formally*

$$y \equiv x^2 \pmod{N}.$$

Victor wishes to see whether Peggy's claim is true, so he asks Peggy to join his interactive procedure:

- *Peggy chooses a random number r modulo N, then send to Victor the number*

$$s \equiv r^2 \pmod{N}.$$

- *Victor randomly chooses a number $\beta \in \{0, 1\}$ and sends β to Peggy.*
- *Peggy computes and sends to Victor the number*

$$z \equiv \begin{cases} r \pmod{N}, & \textit{if } \beta = 0 \\ xr \pmod{N}, & \textit{if } \beta = 1. \end{cases}$$

- *Victor computes the number* $z^2 \pmod N$ *and check that*

$$z^2 \equiv \begin{cases} s \pmod N, & \text{if } \beta = 0 \\ ys \pmod N, & \text{if } \beta = 1. \end{cases}$$

  If this is true, Victor accepts Peggy's response as being valid; otherwise, Victor rejects it.

Peggy and Victor repeat the procedure n times, where n is reasonably large, about $n = 80$. If all of Peggy's responses are acceptable, then Victor accepts Peggy's claim that she knows a number y that is a square modulo N; otherwise, he rejects Peggy's claim.

To have a better illustration of the interactive procedure in the protocol, we use the sequence diagram shown in Figure 2.2.

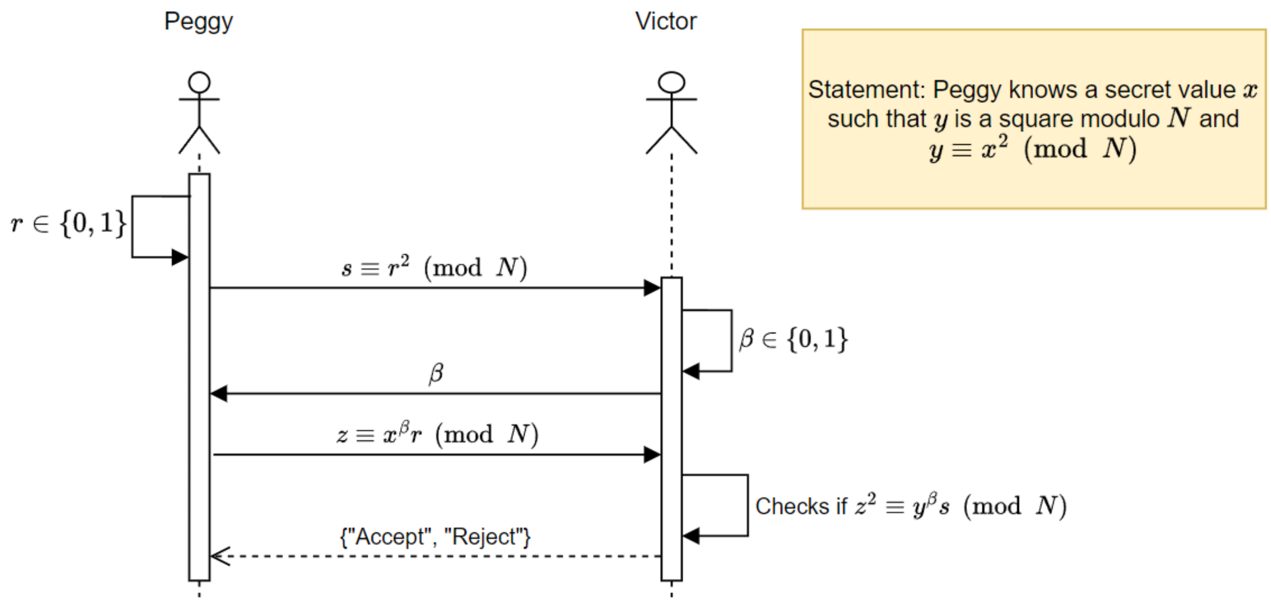

Peggy
Victor
Statement: Peggy knows a secret value x such that y is a square modulo N and $y \equiv x^2 \pmod N$
$r \in \{0,1\}$
$s \equiv r^2 \pmod N$
$\beta \in \{0,1\}$
β
$z \equiv x^\beta r \pmod N$
Checks if $z^2 \equiv y^\beta s \pmod N$
{"Accept", "Reject"}

Figure 2.2: *The sequence diagram for the interactive procedure in Example 2.3.2.*

We now express some mathematical notation to help define the formal definition of a zero-knowledge proof. Let $\mathcal{X}$ be the set of statements, $\mathcal{W}$ be the set of witnesses, and let $\mathcal{R} \subseteq \mathcal{X} \times \mathcal{W}$ be a **relation**. The prover has to prove to the verifier that he knows a secret witness $w \in \mathcal{W}$ that satisfies $(x, w) \in \mathcal{R}$ for a statement $x \in \mathcal{X} \in \{0,1\}^*$ without revealing any information about w, while the verifier learns nothing other than the fact that $(x, w) \in \mathcal{R}$. With this intuition, we now describe the properties of a zero-knowledge proof.

Definition 2.3.3 (Zero-knowledge Proofs Properties)**.** Let $\mathsf{ZKP}(x)\langle P(w), V\rangle \in \{0,1\}$ be a zero-knowledge protocol between the prover P and the verifier V with a statement x. Then $\mathsf{ZKP}(x)\langle P(w), V\rangle$ satisfy the following properties:

- **Completeness**. If the statement x holds for some secret witness w, then the protocol always outputs 1. Formally speaking

$$\exists w : (x, w) \in \mathcal{R} \Rightarrow \Pr[\mathsf{ZKP}(x)\langle P(w), V\rangle = 1] = 1.$$

- **Soundness**. If the statement x does not hold for all secret witnesses w, then there is a negligible probability that the protocol outputs 1. Formally speaking

$$(x, w) \notin \mathcal{R} \text{ for all } w \in \mathcal{W} \Rightarrow \Pr[\mathsf{ZKP}(x)\langle P(w), V\rangle = 1] \leq \mathsf{negl}(|x|).$$

- **Zero-knowledge**. If the statement x holds for some secret witness w, then the verifier V learns nothing other than the fact that the statement x holds for some secret witness w. Formally speaking, for all $(x, w) \in \mathcal{R}$ and adversaries $\mathcal{A}$ with polynomial running time in $|x|$, there exists a simulator $\mathcal{S}$ such that

$$\left|\Pr\left[\ \mathcal{A}(\mathsf{tr}) = 1 \ \middle|\ \mathsf{tr} \leftarrow \mathsf{View}(\mathsf{ZKP}(x)\langle P(w), V\rangle\ \right] - \Pr\left[\ \mathcal{A}(\mathsf{tr}) = 1 \ \middle|\ \mathsf{tr} \leftarrow \mathcal{S}(x)\ \right]\right|$$

is negligible in $|x|$, where $\mathsf{View}(\mathsf{ZKP}(x)\langle P(w), V\rangle)$ outputs the public transcript made by P and V during the protocol. Informally speaking, the simulator $\mathcal{S}$ has to create an accepting transcript tr without the help of the prover with the secret witness w such that it is computationally indistinguishable from the actual transcript created by P and V in the protocol. The verifier can not learn anything about the witness w from the transcript tr since tr is created without any information about w. Note that $\mathcal{S}$ does not need to accurately follow the protocol, as long as $\mathcal{S}$ creates an accepting transcript tr that matches the actual transcript.

We now provide proofs for completeness, soundness, and zero-knowledge of the zero-knowledge protocol described in Example 2.3.2 as follows:
Completeness. If y is indeed a square modulo N, then

$$z^2 \equiv (x^\beta r)^2 \equiv x^{2\beta} r^2 \equiv y^\beta s \pmod{N}.$$

Therefore, Victor always accepts Peggy's response.

Soundness. If y is not a square modulo N , then regardless of how Peggy chooses s, only s or ys is a square modulo N. Hence there is a 50% chance that Peggy fails to answer Victor's challenge. By repeating the procedure n times ($n = 80$ for example), Peggy has a $2^{-n} = 2^{-80}$ chance to cheat the procedure (which is negligible for reasonably large n).

Zero-knowledge. Suppose Victor has done interacting with Peggy and now he wants to prove to another verifier Veronica that y is a square modulo N. When the communication between Victor and Veronica ends, he creates a list of tuples

$$(s_1, \beta_1, z_1), (s_2, \beta_2, z_2), \cdots ,$$

where each tuple satisfies

$$z_i^2 \equiv y^{\beta_i} s_i \pmod N.$$

If we consider Victor to be the simulator $\mathcal{S}$, then the list of tuples he creates is the transcript tr. Moreover, without knowing the witness x, he can still create the same transcript that can be proved to have an indistinguishable distribution from the actual transcript of the protocol executed by Peggy and Victor. Victor creates the same accepting transcript without knowing x by randomly chooses $z \xleftarrow{\$} \mathbb{Z}_N^*$ and $\beta \xleftarrow{\$} \{0; 1\}$, then simply computes $s \equiv z^2(y^\beta)^{-1} \pmod N$. Note that $\mathcal{S}$ does not have to follow the execution of the protocol to produce the same accepting transcript. For example, Victor does not have to compute s first and then compute z, as long as the triple (z, β, s) matches the actual transcript at the end of the protocol. When the created transcript matches the distribution of the actual transcript, it indicates that Victor learns nothing from the interaction between him and Peggy, because if he did, then the distributions would be different.

We will explain the concept of arithmetization used in any zero-knowledge proof system. We can say that the $\mathcal{R}$ specifies combinations of public inputs and private inputs (the witnesses) that are valid, and the implementation of $\mathcal{R}$ to be used in a specific proof system is called a **circuit**. Also, the language we use to express circuits to a particular proof system is called an **arithmetization**. Normally, an arithmetization will define circuits in terms of *polynomial constraints* on variables over a field. To create a proof for a statement, the prover needs to know the private inputs and the intermediate values of the circuit (called **advice** values), which depends on how we write the circuit.

Example 2.3.3. *Suppose we want to build a circuit that can prove in zero-knowledge that we know a preimage x of a given hash function H for a digest $y = H(x)$. Then we need to know:*

- *The private inputs: the preimage x.*
- *The public inputs: the digest y.*
- *The relation $\mathcal{R} = \{(x, y) : y = H(x)\}$.*
- *The advice values: all the intermediate values in the circuit that implement the hash function H. In this case, the witness will be x.*

2.3.5 Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK)

A **succinct non-interactive argument of knowledge (SNARK)** is a proof system which satisfies the "Completeness", "Soundness" described in section 2.3.4 and the following additional properties:

- **Knowledge soundness**. For any prover able to produce a valid proof with probability ϵ, there exists an extractor capable of extracting a witness for said statement with probability ϵ [Nit20]. Note that this property is different from the "soundness" property described in Section 2.3.4, since "knowledge soundness‘ holds for all statements x, whereas "soundness" only holds for all statements $x \in \mathcal{X}$.
- **Non-interactivity**. SNARKs do not require rounds of interaction between the prover and the verifier.
- **Succinctness**. The size of the proof is very small compared with the size of the witness, i.e., the size of the computation itself. Specifically, the length of the proof is

  $$|\pi| = poly(\lambda)polylog(|x| + |w|)$$

  with x be the instance value and w be the witness value [Gro16].

A **zero-knowledge SNARK (zk-SNARK)** is a SNARK that achieves *zero-knowledge* property described in Section 2.3.4. zk-SNARKs are widely used in many practical applications. For example, Zcash[1], Mina[2], Hawk [KMS+15], Pinocchio [PGHR13], etc. use zk-SNARKs to enhance privacy in their systems and solve their scalability problems. Most zk-SNARK constructions utilize *arithmetization* to transform computational statements into algebraic statements, which consist of polynomials over a finite field. Hence, the zero-knowledge proof is computed based on the output of arithmetization, and an *arithmetic circuit* is an endpoint of arithmetization. Arithmetic circuits share similarities to Boolean circuits, with the exception that the wire values are integers instead of bits, and instead of using Boolean operations (AND, OR, NOT, etc), arithmetic circuits use arithmetic operations (addition, multiplication, etc). Circuit computations are better for unstructured computations and support composability with relative ease.

Available zk-SNARK constructions employ the *common reference string model* [Dam00]. The common reference string (CRS) model captures the assumption that a trusted setup in which all involved parties get access to the same string s taken from some distribution D exists. SNARKs can be categorized based on the need for a trusted setup as follows:

- *Preprocessing SNARKs*: These are SNARKs that must require trusted setup initialization before proof generation can be done. Some of the SNARKs of this type are Plonk [GWC19], Marlin [CGS+23], Sonic [MBKM19], etc.

- *Transparent SNARKs*: These are SNARKs that do not require trusted setup initialization before proof generation can be done. Some of the SNARKs of this type are STARK [BSBHR18], Aurora [BSCR+18], Halo [BGH19], etc.

Considering the use of the CRS model in constructions, SNARKs can be classified into the following types:

- *Non-universal SNARKs*: For each specific circuit to be proved, a common reference string must be generated for said circuit. An example of non-universal SNARKs [BCG+13] is Groth16 [Gro16].

- *Universal SNARKs*: These are SNARKs that only require one universal common reference string to generate proofs for various circuits. Examples of universal SNARKs are Plonk [GWC19], Marlin [CGS+23], Sonic [MBKM19], etc.

[1]`https://z.cash/`
[2]`https://minaprotocol.com/`

2.3.6 Commitment Schemes

Commitment schemes are fundamental concepts for constructing many cryptographic protocols, including zero-knowledge proofs and the field of Multi-Party Computation (MPC). In this section, we formally define a commitment scheme, then we list some commitment schemes that will be used in Chapter 5 such as polynomial commitment schemes and the Merkle Proof-of-inclusion for Merkle Tree.

2.3.6.1 Definition

We now formally define commitment schemes and their basic properties by following the definitions in [JKPT12]:

Definition 2.3.4 (Commitment Scheme)**.** A **commitment scheme** consists of 3 polynomial-runtime algorithms: Setup, Commit, and Verify, described as follows:

- $\mathsf{pk} \leftarrow \mathsf{Setup}(1^\lambda)$: A setup algorithm that takes some bitstring 1^λ as an input, where λ is the security parameter, and outputs some public parameter pk that is used for the rest of the algorithms in the scheme.
- $(c, d) \leftarrow \mathsf{Commit}(\mathsf{pk}, m)$: A commitment algorithm that takes the public key pk and a message m to commit, and outputs a commitment c together with some decommitment information d.
- $\{0, 1\} \leftarrow \mathsf{Verify}(\mathsf{pk}, m, c, d)$: A verification algorithm that takes the public key pk, the message that is opened by the committer m, a commitment c and decommitment information d of the message m as inputs. The algorithm outputs whether the commitment is accepted (1), or rejected (0).

A good commitment scheme should satisfy these two properties: **binding** and **hiding**. A commitment scheme is called **hiding** if no information about the committed value is revealed after the commitment stage. A commitment scheme is called **binding** if it is hard for an adversary to come up with two different values and generate the same valid commitment. We formally defines these two properties below:

Definition 2.3.5 (Binding Property)**.** A commitment scheme is called **computational binding** if it is computationally infeasible for an adversary to generate two different values whose commitment values are identical. Formally, there exists a negligible function negl such that

$$\Pr\left[\ \mathsf{Commit}(\mathsf{pk}, m') = \mathsf{Commit}(\mathsf{pk}, m) = c \ \middle|\ (m, m' \neq m) \leftarrow \mathcal{A}(\lambda, \mathsf{pk})\ \right] = \mathsf{negl}(\lambda).$$

If the adversary has unlimited computational power, then the commitment scheme is called **perfect binding**.

Definition 2.3.6 (Hiding Property)**.** A commitment scheme is called **hiding** if for every messages $m, m' \in \mathcal{M}$ and $(c, d) \leftarrow \mathsf{Commit}(\mathsf{pk}, m)$, $(c', d') \leftarrow \mathsf{Commit}(\mathsf{pk}, m')$, the probability distribution of c and c' is computationally indistinguishable over the choice of $\mathsf{pk} \overset{\$}{\leftarrow} \mathsf{Setup}(1^\lambda)$. Informally, A commitment scheme is called hiding if receiving the commitment c reveals no information about the committed value m.

2.3.6.2 Polynomial Commitment Schemes

In this section, we define a polynomial commitment scheme by following the definition in [KZG10]:

Definition 2.3.7 (Polynomial Commitment Schemes)**.** A **polynomial commitment scheme** consists of 6 algorithms, namely Setup, Commit, Open, $\mathsf{VerifyPoly}$, $\mathsf{CreateWitness}$, and $\mathsf{VerifyEval}$, described as follows:

$\mathsf{pk} \leftarrow \mathsf{Setup}(1^\lambda, t)$: Generates an appropriate algebraic structure $\mathcal{G}$ and a public-private key pair $(\mathsf{pk}, \mathsf{sk})$ to commit to a polynomial of degree $\leq t$. The algorithm outputs pk and deletes sk. Note that Setup is run by a trusted or distributed authority, and the secret key sk is not required in the rest of the scheme.

$(c, d) \leftarrow \mathsf{Commit}(\mathsf{pk}, \phi(x))$: Outputs a commitment $\mathcal{C}$ to a polynomial $\phi(x)$ for the public key pk, together with some decommitment information d. Note that in some constructions, d may be null.

$(\phi(x), d) \leftarrow \mathsf{Open}(\mathsf{pk}, c, \phi(x), d)$: Outputs the polynomial $\phi(x)$ used while creating the commitment, with decommitment information d.

$\{0, 1\} \leftarrow \mathsf{VerifyPoly}(\mathsf{pk}, c, \phi(x), d)$: Verifies that $\mathcal{C}$ **is a commitment to** $\phi(x)$. If so, it outputs 1, otherwise it outputs 0.

$(i, \phi(i), w_i) \leftarrow \mathsf{CreateWitness}(\mathsf{pk}, \phi(x), i, d)$: Outputs a witness for the evaluation $\phi(i)$ of $\phi(x)$ at index i and a decommitment information d.

$\{0,1\} \leftarrow \mathsf{VerifyEval}(\mathsf{pk}, \mathcal{C}, i, \phi(i), w_i)$: Verifies that $\phi(i)$ **is the evaluation at the index** i **of** $\phi(x)$, where $\mathcal{C}$ is the commitment of $\phi(x)$. If so, it outputs 1, otherwise it outputs 0.

In terms of security, a malicious committer should not be able to convincingly present two different values as evaluations $\phi(i)$ that have the same commitment $\mathcal{C}$. Also, if less than $deg(\phi)$ points of $\phi(x)$ are revealed, then the polynomial should remain hidden. We formally define the security and correctness of a polynomial commitment scheme as follows

Definition 2.3.8 (Security and Correctness of Polynomial Commitment Schemes)**.** $(\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{VerifyPoly}, \mathsf{CreateWitness}, \mathsf{VerifyEval})$ is a secure polynomial commitment scheme if it satisfies all the following properties:

Correctness. Let $\mathsf{pk} \leftarrow \mathsf{Setup}(1^\lambda)$ and $\mathcal{C} \leftarrow \mathsf{Commit}(\mathsf{pk}, \phi(x))$. For all $\phi(x) \in \mathbb{Z}_p[x]$:

- The output of $\mathsf{Open}(\mathsf{pk}, \mathcal{C}, \phi(x))$ is successfully verified by $\mathsf{VerifyPoly}(\mathsf{pk}, \mathcal{C}, \phi(x))$.
- Any output $(i, \phi(i), w_i)$ of $\mathsf{CreateWitness}(\mathsf{pk}, \phi(x), i)$ is successfully verified by $\mathsf{VerifyEval}(\mathsf{pk}, \mathcal{C}, i, \phi(i), w_i)$.

Polynomial Binding. For all adversaries $\mathcal{A}$ with polynomial runtime in λ, it holds that

$$\Pr\left[\begin{array}{l|l} (\mathcal{C}, \phi(x), \phi'(x)) \leftarrow \mathcal{A}(\mathsf{pk}, \lambda) & \\ \phi(x) \neq \phi'(x) & \\ \mathsf{VerifyPoly}(\mathsf{pk}, \mathcal{C}, \phi(x)) = 1 \wedge & \mathsf{pk} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{VerifyPoly}(\mathsf{pk}, \mathcal{C}, \phi'(x)) = 1 & \end{array}\right] = \mathsf{negl}(\lambda).$$

Evaluation Binding. For all adversaries $\mathcal{A}$ with polynomial runtime in λ, it holds that

$$\Pr\left[\begin{array}{l|l} (\mathcal{C}, (i, \phi(i), w_i), (i, \phi(i)', w_i')) \leftarrow \mathcal{A}(\mathsf{pk}, \lambda) & \\ \phi(i) \neq \phi(i)' & \\ \mathsf{VerifyEval}(\mathsf{pk}, \mathcal{C}, i, \phi(x), w_i) = 1 \wedge & \mathsf{pk} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{VerifyEval}(\mathsf{pk}, \mathcal{C}, i, \phi'(i), w_i') = 1 & \end{array}\right] = \mathsf{negl}(\lambda).$$

Hiding. Given $(\mathsf{pk}, \mathcal{C})$ and $\{(i_j, \phi(i_j)), w_{\phi_{i_j}} : j \in [1, deg(\phi)]\}$ for a polynomial $\phi(x) \overset{\$}{\leftarrow} \mathbb{Z}_p[x]$ such that for each j, $\mathsf{VerifyEval}(\mathsf{pk}, \mathcal{C}, i_j, \phi(i_j), w_{\phi_{i_j}}) = 1$:

- No adversary $\mathcal{A}$ can determine $\phi(\hat{i})$ with non-negligible probability for any *unqueried index* $\hat{i}$ (also called **computational hiding**).
- No computationally unbounded adversary $\mathcal{A}$ has any information about $\phi(\hat{i})$ given any *unqueried index* $\hat{i}$ (also called **unconditional hiding**).

2.4 Merkle Tree

In this section, we define a Merkle Tree and explain the concepts of Merkle proof-of-inclusions and multiproofs, which are used in our proposed solution in committing the execution trace of a RAM program in Chapter 5. Most of the contents are followed from [TK22] and [Car09].

2.4.1 Definition

A **Merkle tree** is a binary tree where each internal node stores the hash value of the children's nodes, and the leaves of the Merkle tree are the hash values of the data in the tree. Figure 2.3 gives an example of a Merkle tree with 8 data elements $D_i : i = 1..8$ as 8 leaves, and H_i's are the outputs of some cryptographic hash function with D_i as inputs. The hash value of an internal node is determined by concatenating two hash values of its children's nodes. We compute until we reach the root, which is called the **Merkle root**.

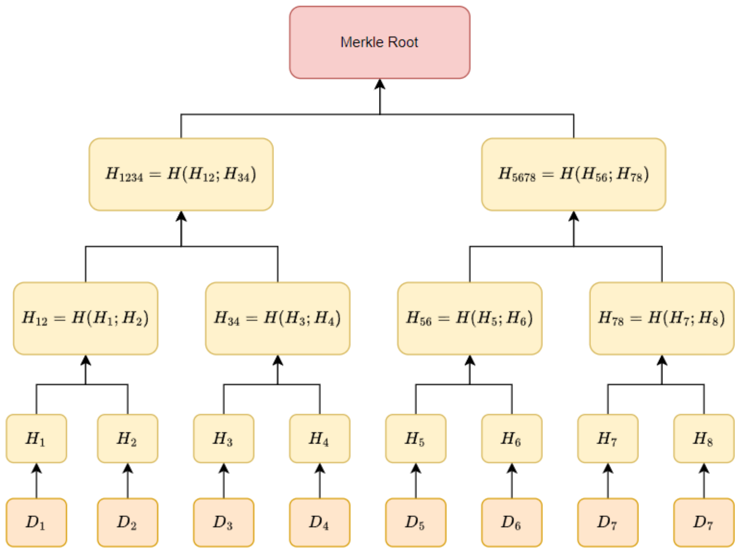


Figure 2.3: *An example of a Merkle Tree with 8 data elements*

2.4.2 Merkle Proof-of-Inclusions

A **Merkle Proof-of-inclusion** is a proof for the statement: "A data element D_i is included in the Merkle Tree". To form a Merkle proof, we need 3 main components described as follows:

- The hash of the data we want to prove.
- The correct Merkle Root.
- Sibling hashes along the path from the leaf to the Merkle root.

Since creating a Merkle proof includes traversing from a leaf to the Merkle root, the time complexity for creating the Merkle proof is $\mathcal{O}(\log_2 n)$.

Example 2.4.1. *Suppose we want to create a Merkle proof certifying that D_4 belongs to the Merkle Tree. The proof consists of the following components:*

- *The hash of the data we want to prove: this will be H_4.*
- *The correct Merkle root in the Merkle tree (denoted by H_{ROOT}).*
- *The sibling hashes along the path from D_4 to H_{ROOT}: The path from D_4 to H_{ROOT} is: $D_4 \longrightarrow H_4 \longrightarrow H_{34} \longrightarrow H_{1234} \longrightarrow H_{ROOT}$. Along the path, we need sibling hashes to be able to compute the Merkle root: H_3 is needed to compute H_{34}, H_{12} is needed to compute H_{1234}, and H_{5678} is needed to compute H_{ROOT}.*

Hence, the Merkle proof of D_4 is $\pi = \{H_4, H_3, H_{12}, H_{5678}, H_{ROOT}\}$, which is also illustrated in Figure 2.4.

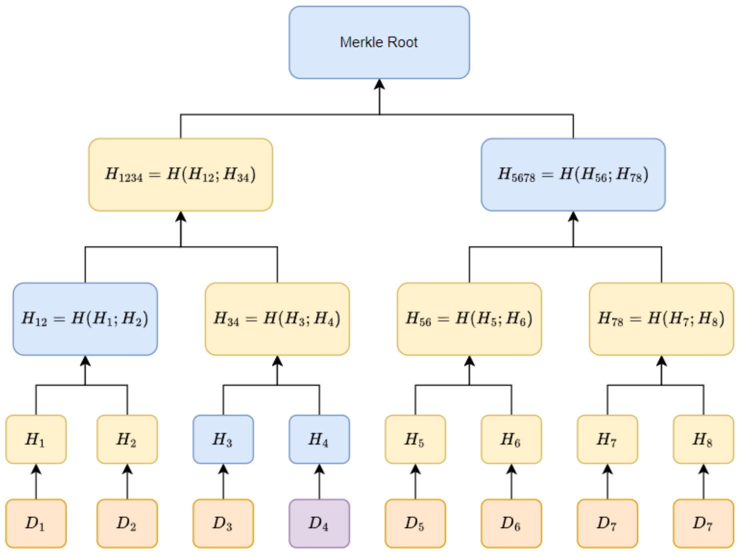

Merkle Root
$H_{1234} = H(H_{12}; H_{34})$
$H_{5678} = H(H_{56}; H_{78})$
$H_{12} = H(H_1; H_2)$
$H_{34} = H(H_3; H_4)$
$H_{56} = H(H_5; H_6)$
$H_{78} = H(H_7; H_8)$
H_1 H_2 H_3 H_4 H_5 H_6 H_7 H_8
D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_7

Figure 2.4: *Merkle proof-of-inclusion for D_4.*

2.4.3 Merkle Multiproofs

We can extend Merkle proofs to not just prove that one piece of data is in the Merkle Tree, but to prove that a set of data elements are in the Merkle Tree. To form a Merkle inclusion proof for a data subset of $\{D_i : 1 \leq i \leq n\}$, where n is the number of leaves in the Merkle Tree, we perform the following steps:

- Create a Merkle proof for a single data element in the subset.
- Combine all Merkle proofs of a single data element to form a single Merkle proof π of the data subset.
- Remove any abundant nodes in the Merkle proof to reduce the size of the proof.

Example 2.4.2. *From the Merkle tree in Figure 2.3, we will compute the Merkle inclusion proof for data elements D_1, D_4 and D_7.*

1. *Firstly, we compute a Merkle proof for D_1, for D_4 and D_7.*
   - $\pi_1 = \{H_1, H_2, H_{34}, H_{5678}, H_{ROOT}\}$.
   - $\pi_4 = \{H_4, H_3, H_{12}, H_{5678}, H_{ROOT}\}$.

- $\pi_7 = \{H_7, H_8, H_{56}, H_{1234}, H_{ROOT}\}$.

2. *We combine all the Merkle proofs π_1, π_4, π_7 to form a single proof π:*

$$\pi = \pi_1 \cup \pi_4 \cup \pi_7 = \{H_1, H_2, H_3, H_4, H_7, H_8, H_{12}, H_{34}, H_{56}, H_{1234}, H_{5678}, H_{ROOT}\}.$$

3. *We remove any unnecessary nodes in π: Since we can compute H_{12} from H_1 and H_2, we do not need H_{12} in the proof. Similarly, we remove H_{34}, H_{1234} and H_{5678}.*

Finally, we complete the Merkle multiproof for the inclusion of $\{D_1, D_4, D_7\}$:

$$\pi = \{H_1, H_2, H_3, H_4, H_7, H_8, H_{56}, H_{ROOT}\}.$$

The Merkle proof in this example is also illustrated in Figure 2.5.

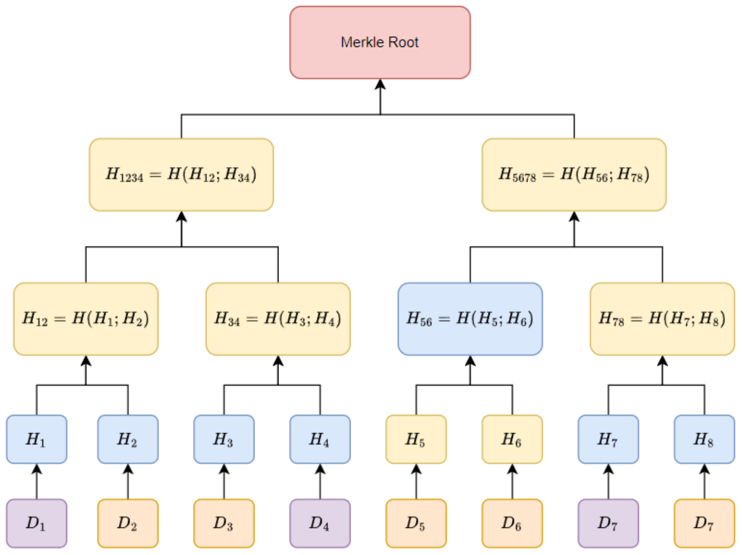


Figure 2.5: *Merkle multiproof for D_1, D_4, D_7.*

2.5 RAM Programs

In this section, we introduce basic concepts of RAM programs by following the definitions from [FKL+21] and [dSGOTV22].

Definition 2.5.1 (RAM Programs)**.** A **RAM program** is defined by a "next-instruction circuit" Π that, given its current state st and a value d (which is equal to the *last-read element*), outputs the next instruction to execute together with the updated state st'. If a memory M is initialized to $M_0, ..., M_{N-1}$ and $M_i \in \{0,1\}^W$, then an execution of RAM program with memory M proceeds as follows:

- Set $\mathsf{st} := \mathsf{start}$ and $d := 0^W$.
- Do until termination:
  - Compute the 4-tuple $(op, l, d', \mathsf{st}') := \Pi(\mathsf{st}, d)$ and set $\mathsf{st} := \mathsf{st}'$.
  - Consider the operation op:
    * If $op = \mathsf{Stop}$, terminates with output d'.
    * If $op = \mathsf{Read}$, set $d := M_l$.
    * If $op = \mathsf{Write}$, set $M_l := d'$ and $d := d'$.

Note that we assume the number of entries N and the entry bit size W to be hard-coded in Π.

Π can be a specific algorithm such as sorting algorithms executing in the RAM model. Π can also be a general-purpose CPU that can execute arbitrary assembly code loaded into a portion of memory. If so, we can design the state st of Π to have an additional *program counter* or an array of *registers* R. We can also design a portion of the memory M to specifically hold the executed program (assembly code), called *program memory* M_p, and the rest to be the *main memory* M_d. One can regard the *program memory* to be read-only. The specific details of designing the memory for the RAM program are explained in Section 5. Note that the RAM program is different than the RAM architecture for a general computer, so it does not relate to any issue of security.

Complexity analysis. The *time complexity* of a RAM program is the number of instructions considered in the above execution. The *space complexity* of a RAM program is the maximum number of entries used by the memory M during execution.

Chapter 3

Related Works

In recent years, there have been many attempts to solve the problem of proving a valid and correct execution of the RAM program. Both these attempts have their advantages and disadvantages. In this section, we briefly review early and recent attempts to work in RAM programs and summarise the advantages and drawbacks of their attempts. We also give a quick literature survey on the available research and implementations in zero-knowledge virtual machines (which are virtual machines that execute programs and output proof of their correct execution in zero-knowledge).

3.1 Literature survey on the implementations of Zero-Knowledge Virtual Machines

Recall that a *zero-knowledge virtual machine (zkVM)* is a virtual machine that executes a program on a specific computation model and outputs the proof that it executes correctly in zero-knowledge (without revealing any internal activities that occur in the machine). zkVMs are one direct instance of proving the correct execution in zero-knowledge, with many applications spanning various fields, including secure data sharing, private smart contracts, and confidential transactions in blockchain networks. There have been numerous research and projects for zkVMs in recent years. The Privacy and Scaling Exploration[1] (or PSE) is a multidisciplinary team that focuses on researching and developing technologies to enhance both privacy and scalability on the Ethereum network. Specifically, they work on solutions based on zero-knowledge proofs and cryptographic techniques to ensure that transactions and

[1]https://pse.dev

smart contract executions can be performed privately and efficiently. One of their projects includes the Rust implementation of *zero-knowledge Ethereum Virtual Machines*[2] *(zkEVMs)*, which is the zkVM that is specifically compatible to the Ethereum infrastructure. Risc Zero[3] is a startup providing solutions on zkVMs that are specifically compatible with the RISC-V platform[4]. Cairo [GPR21] is a programming language developed by StarkWare and designed specifically for writing decentralized applications (DApps) on StarkNet. Drawing inspiration from the Rust programming language, Cairo implementation[5] features a set of capabilities that enable secure, efficient, and scalable code development. Finally, Orochi Network[6] is a decentralized oracle network designed to provide secure, reliable, and scalable off-chain data to smart contracts and decentralized applications (dApps) on various blockchain platforms. It aims to solve the oracle problem by ensuring data integrity, reducing latency, and enhancing the overall efficiency of blockchain ecosystems, using zero-knowledge proofs and many cryptographic primitives. One of many solutions is the project called zkMemory[7], which is a universal memory commitment module written in Rust programming language. We will explain this particular project in greater detail in Chapter 7.

3.2 Related research on zero-knowledge proofs for RAM programs

In 2015, Hu, Mohassel, and Rosulek [HMR15] proposed a zero-knowledge proof system in the RAM computation model in which the addresses of memory accesses are committed by the prover and can later be revealed to the verifier. They applied the concept of *oblivious RAM (or ORAM)*, which was first introduced by Goldreich and Ostrosvsky [GO96], to achieve this feature. ORAM provides a wrapper to encode a (possibly large) logical dataset as a physical dataset and translates each logical memory access into a sequence of physical memory accesses such that the physical memory access patterns reveal no information about the underlying logical memory accesses. Their goal was to achieve sublinear computation and communication time to the memory size (or $|M|$) in their construction. Subsequently, Mohassel, Rosulek, and Scafuro [MRS17] proposed a solution similar to [HMR15], with the

[2] `https://github.com/privacy-scaling-explorations/zkevm-circuits`
[3] `https://dev.risczero.com/api/zkvm/`
[4] See their project *risc0*: `https://github.com/risc0/risc0`
[5] `https://github.com/starkware-libs/cairo`
[6] `https://orochi.network/`
[7] `https://github.com/orochi-network/orochimaru/tree/main/zkmemory`

exception that in their construction, the overhead of the verifier in any initialization phase did not depend on the $|M|$. Both of these constructions had one limitation: the cost of the prover in the initial commitment of the memory scales with $|M|$, while $|M|$ in a RAM program can be very large. Moreover, to the best of our knowledge, neither of these constructions has been implemented.

Constructions proposed by [BCG+13], [BCTV14], and [WSR+15] avoided the use of ORAM. Instead, they used the TinyRAM framework which used the sorting network to prove the consistency of memory accesses. TinyRAM was first proposed by [BSCG+20]. Following their definition, TinyRAM is essentially a Reduced Instruction Set Computer (RISC) with byte-level addressable random-access memory. TinyRAM instruction set only contains 29 instructions which strikes the balance between *sufficient expressibility* (i.e. supports efficient and short assembly code when compiled from high-level programming language) and *small instruction set*. Using the sorting network from their construction, the output circuit is smaller in size compared to naive arithmetic circuit implementation. A later construction by [BHR+20] proposed a space-efficient commitment scheme for multilinear polynomials, which reduced the overhead of the TinyRAM protocol. However, its computational efficiency was not concretely shown in the paper. Bootle et al. [BCG+18] reduced the verifier computation time to sublinear asymptotic performance ($\mathcal{O}(\alpha T)$ TinyRAM steps). However, to the best of our knowledge, their construction had not been implemented, and the constant α in the verifier computation complexity was very large, which was not recommended for implementation, as the author suggested.

David Heath and Vladimir Kolesnikov developed BubbleRAM [HK22], which integrated the sorting network approach with *garbled-circuit zero-knowledge protocols* described in [JKO13, HK20, FNO14]. The garbled circuit is a cryptographic protocol that allows secure computation between two mistrusting parties, and two mistrusting parties can jointly evaluate a function over all of their private inputs without the trusted third party. BubbleRAM allowed for zero-knowledge proofs about the execution of a lightweight, general-purpose CPU, which required $\mathcal{O}(\log^2 n)$ communication cost per memory access. However, prior works [BSCTV14, BCTV14, BCG+13] showed constructions that were non-interactive and succinct, which, in exchange for their performance, were applicable to more problems than BubbleRAM. They also developed BubbleCache [HYDK22], which also integrated the sorting network approach with garbled-circuit zero-knowledge protocols, with the exception that they employed *multi-level caching*, which reduced the communication cost per access from $\mathcal{O}(\log_2 n)$ to $\mathcal{O}(\log n)$. However, one small limitation is that their construction also introduced a possibility of up to 30% cache miss rate.

Franzese et al. [FKL+21] proposed a *constant-overhead* interactive stateful zero-knowledge proof system for RAM programs. The term *constant-overhead* means that the communication complexity and both the prover and the verifier running times complexity scales linearly with the size of the memory M and the running time T of the underlying RAM program. The term *stateful* means that the prover can commit to values and then repeatedly use those committed values for different zero-knowledge proofs. Their approach was using a *polynomial-based equality check* to ensure the consistency of memory accesses. Their construction yields concrete efficiency improvement to prior constructions mentioned above. However, their work had several limitations: First, their construction required a trusted setup (i.e., the prover and the verifier send their data to the trusted setup to process before receiving the data to use for different zero-knowledge proofs); Second, their construction worked on designated verifier (i.e. only one verifier holds the global authentication key). Meanwhile, Guilhem et al. [dSGOTV22] proposed a constant-overhead *stateless* zero-knowledge proof system for RAM programs. The key feature of their work was that they used an arithmetic circuit C_{check} to check the consistency of a list of memory access tuples in zero knowledge. Their work also achieved concrete efficiency improvement that both the communication cost and the runtimes of the prover and verifier asymptotically scale linearly with the size of the memory and the run time of the underlying RAM program. Their work was also implemented using C++ with concrete performance results. However, they shared the same limitation with Franzese et al. [FKL+21] that they also required a trusted setup. Also, their performance tests, compared to other works, did not take into account the communication time between parties.

Chapter 4

An Overview of Halo2 Protocol

This chapter presents technical details of the Halo2 framework, a zero-knowledge proof protocol based on PLONK, and all other related cryptographic primitives. All this knowledge is the backbone for the implementation described in Chapter 7. Most of the content in this chapter is referenced by the Halo2 documentation website[1], the Halo2 paper [BGH19], and the documentation of Orochi Network[2].

4.1 Background

Halo2 is a SNARK protocol proposed by Zcash [BGH19], a privacy-centric cryptocurrency that is based on Bitcoin's codebase. The arithmetization of Halo2 comes from **PLONK**, a general-purpose zero-knowledge proof scheme devised by Ariel Gabizon, Zac Williamson, and Oana Ciobotaru [GWC19]. Besides its use in ZCash, Halo2 has been used by prominent teams such as Protocol Labs[3], the Ethereum Foundation's Privacy and Scaling Explorations (PSE) team[4], Scroll[5], and Taiko[6], making it one of the most popular zk-SNARK constructions nowadays.

Its former version (which is Halo) was the first practical realization of a recursive proof system without a trusted setup [BGH19]. There are, however, some differences between the

[1]https://zcash.github.io/halo2/index.html
[2]https://docs.orochi.network/
[3]https://filecoin.io/blog/posts/reviving-halo-2-with-protocol-labs-filecoin-foundation-ethereum-foundation-and-electric-coin-co/
[4]https://github.com/privacy-scaling-explorations/halo2
[5]https://github.com/scroll-tech/halo2
[6]https://github.com/taikoxyz/halo2

two versions. Figure 4.1 compares the overall processes of the Halo and Halo2 proof systems[7]. The Polynomial IOP technology in Sonic is applied by Halo to describe a Recursive Proof Composition algorithm, and the Inner Product argument technology in Bulletproof is also employed to replace the Polynomial Commitment Scheme in the algorithm, the replacement which eliminates the dependence on Trust Setup. Halo2 has been further optimized, mainly in the direction of Polynomial IOP. Researchers have recently discovered the Polynomial IOP Scheme, which is more efficient than Sonic, such as Marlin and PLONK. PLONK was selected because it supports a more flexible circuit design.

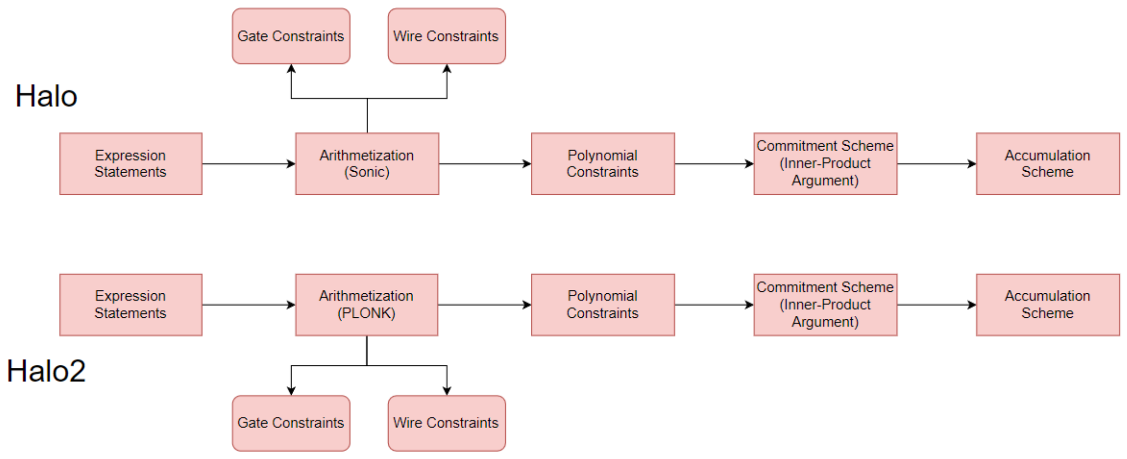


Figure 4.1: *Processes of Halo and Halo2 proof system*

The arithmetization of Halo2 is based on PLONK, specifically its extension called **UltraPLONK**[8] which supports custom gates and lookup arguments. The term **PLONKish** was created by Daira Hopwood[9] to describe variants that are based on PLONK arithmetization.

4.2 Halo2 Arithmetization

This section will explain the arithmetization used in the Halo2 proving system. Before we detail arithmetization in Halo2, we describe the main features of PLONK arithmetization, which is the foundation for Halo2 arithmetization.

[7]https://medium.com/@ola_zkzkvm/halo-principle-explained-fa5a2e2767cd
[8]https://medium.com/aztec-protocol/aztecs-zk-zk-rollup-looking-behind-the-cryptocurtain-2b8af1fca619
[9]https://twitter.com/feministPLT/status/1413815927704014850

4.2.1 PLONK Arithmetization

The PLONK arithmetization was first proposed in the PLONK paper [GWC19] as a method to arithmetize circuits where each gate has at most 2 inputs and 1 output. Each gate is expressed by a *gate constraint* (which is an equation of its inputs and output). The circuit also has *copy constraints* (which are constraints that connect the output of one gate to an input of another gate), and the wiring is represented as permutations. First, we describe the circuit's specifications, then explain how to break the overall circuit into gates and wires. Finally, we present a unified form of gate and handle copy constraints.

4.2.2 Circuit Specification

This section considers arithmetic circuits whose operators are over a finite field $\mathbb{F}$. Let ℓ_{input} denote the number of input wires in the circuit (denoted by $\mathcal{C}$). Assume that $\mathcal{C}$ has precisely n gates. Consider the following possible gates in the circuit:

- *Addition and multiplication gates*: take 2 inputs and return 1 output.
- *Addition and multiplication gates with a constant*: take only 1 input (and a predefined constant) and return 1 output.

Example 4.2.1. *Consider the function $f(x, y) = x^2 + xy + y^2 + 7$. Then the computational sequence can be arranged as follows:*

1. $z_1 = x \cdot x$ *(multiplication).*
2. $z_2 = y \cdot y$ *(multiplication).*
3. $z_3 = x \cdot y$ *(multiplication).*
4. $z_4 = z_1 + z_3$ *(addition).*
5. $z_5 = z_4 + z_2$ *(addition).*
6. $z_6 = z_5 + 7$ *(addition with a constant).*

The input of this circuit is $\ell_{input} = 2$, and the circuit has 6 gates with z_6 being the output of the circuit. The whole desired circuit (including gates and wire labels) is depicted in Figure 4.2 (we will explain the labeling of wires in Section 4.2.3).

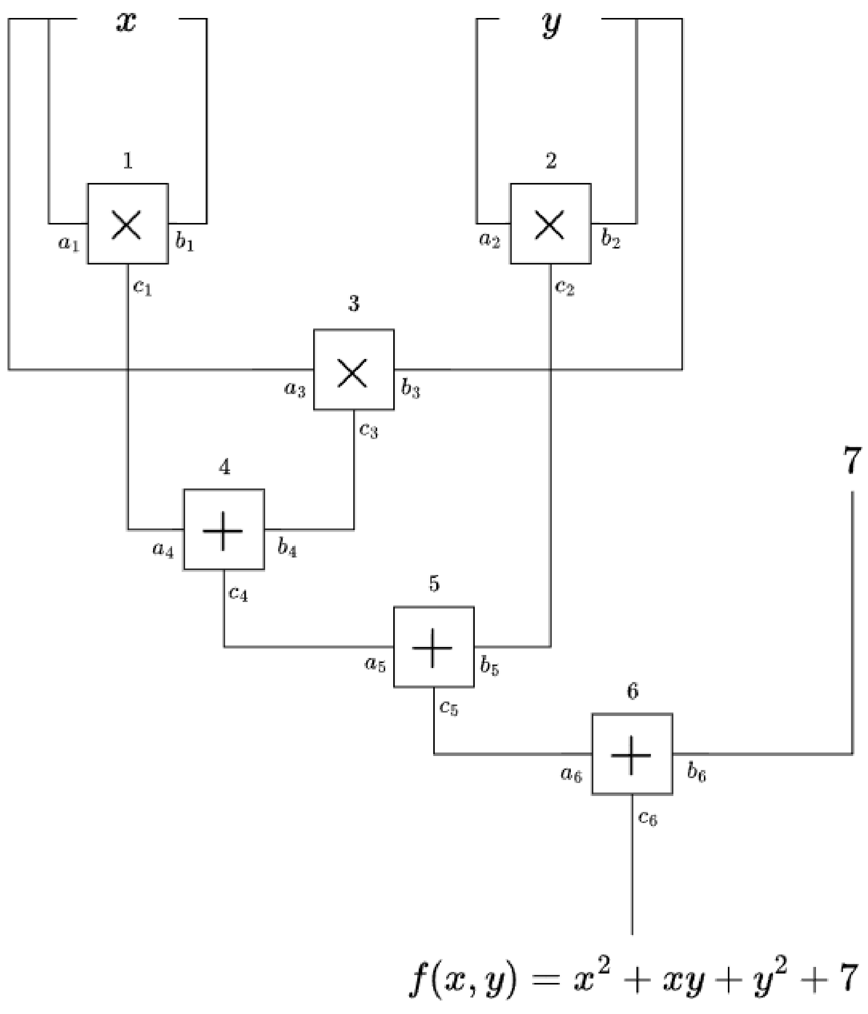


Figure 4.2: *The circuit in Example 4.2.1.*

4.2.3 Breaking the Circuit

This section mostly references the Halo2 documentation website[10]. In order to break the circuit into a set of gates and with wires such that no wire is the input of 2 gates, we use the set of labels $\mathcal{I} = \{a_1, \cdots, a_n, b_1, \cdots, b_n, c_1, \cdots, c_n\}$, where a_i, b_i, c_i are labels of the 2 inputs and the output of the i-th gate in the circuit. Also, let $x : \mathcal{I} \to \mathbb{F}$ be the function that maps the wires to a wire value. Using these notions, $x(i)$ represents the value at wire $i \in \mathcal{I}$. We write x_i instead of $x(i)$ to simplify the notation. For equations in the circuit that ensure the correct computation of the operation, we denote them as **gate constraints**. For equations in the circuit that ensure the equalities among wires in the circuit, we denote them as **wire**

[10]https://zcash.github.io/halo2/index.html

constraints. We will explain more about gate constraints and wire constraints in Section 4.2.6.

Example 4.2.2. *Recall the function we need to build the circuit in Example 4.2.1. By breaking the circuit, we have the following constraints in the above format of gate constraints*

$$\begin{cases} x_{c_1} = x_{a_1} \cdot x_{b_1} \\ x_{c_2} = x_{a_2} \cdot x_{b_2} \\ x_{c_3} = x_{a_3} \cdot x_{b_3} \\ x_{c_4} = x_{a_4} + x_{b_4} \\ x_{c_5} = x_{a_5} \cdot x_{b_5} \\ x_{c_6} = x_{a_6} + 7 \end{cases}$$

and wire constraints:

$$\begin{cases} x = x_{a_1} = x_{b_1} = x_{a_3} \\ y = x_{a_2} = x_{b_2} = x_{b_3} \\ z_1 = x_{c_1} = x_{a_4} \\ z_2 = x_{c_2} = x_{b_5} \\ z_3 = x_{c_3} = x_{b_4} \\ z_4 = x_{c_4} = x_{a_5} \\ z_5 = x_{c_5} = x_{a_6} \\ z_6 = x_{c_6}. \end{cases}$$

4.2.4 Constraints System of the Circuit

To build the constraint system of the circuit, we need to express the gates in the circuit in a unified form. On the gates and wires, we have two types of constraints:

Gate constraints. These constraints are equations between wires attached to the same gate. A single gate with 2 inputs x_a, x_b and 1 output x_c is defined by the equation

$$(q_L)x_a + (q_R)x_b + (q_O)x_c + (q_M)x_a x_b + (q_C) = 0$$

where the value $q = (q_L, q_R, q_O, q_M, q_C)$ is called the **selector vector** and uniquely determined by the corresponding gate, denoted in Table 4.1. Specifically:

- Setting $q = (q_L, q_R, q_O, q_M, q_C) = (1, 1, -1, 0, 0)$ results in the gate constraint being $x_a + x_b = x_c$. This gate is called the **addition gate**.
- Setting $q = (q_L, q_R, q_O, q_M, q_C) = (0, 0, -1, 1, 0)$ results in the gate constraint being $x_a x_b = x_c$. This gate is called the **multiplication gate**.
- Setting $q = (q_L, q_R, q_O, q_M, q_C) = (1, 0, -1, 0, x)$ results in the gate constraint being $x_a + x = x_c$. This gate is called the **addition gate with a constant**.
- Setting $q = (q_L, q_R, q_O, q_M, q_C) = (x, 0, -1, 0, 0)$ results in the gate constraint being $x_a x_b = x_c$. This gate is called the **multiplication gate with a constant**.

Table 4.1: Possible selector vector values for common gate in PLONK arithmetization.

Gate	Selector vector q	Constraint
Addition	$(1, 1, -1, 0, 0)$	$x_a + x_b = x_c$
Multiplication	$(0, 0, -1, 1, 0)$	$x_a x_b = x_c$
Addition with constant	$(1, 0, -1, 0, x)$	$x_a + x = x_c$
Multiplication with constant	(x, 0, -1, 0, 0)	$x_a x = x_c$

Copy constraints. Recall that gate constraints do not ensure equalities of wire values of gates, making the circuit inconsistent between the gates. This can be solved using copy constraints. These are equality constraints between the output of one gate and the input of another gate or between the output of a gate and a public value. Copy constraints ensure the equality of different wires anywhere in the circuit. Consider Example 4.2.1, one of the copy constraints is $z_1 = x_{c_1} = x_{a_4}$, which ensures that the value of wire c_1 (the output of gate 1) is equal to the value of wire a_4 (one input of gate 4), which equals z_1 (the output of gate 1). In short, handling copy constraints in the circuit require that there exists a permutation $\sigma : \mathcal{I} \to \mathcal{I}$ such that

$$((a_1, x_{a_1}), \cdots, (a_n, x_{a_n}), (b_1, x_{b_1}), \cdots, (b_n, x_{b_n}), (c_1, x_{c_1}), \cdots, (c_n, x_{c_n}))$$

is a permutation of

$$((\sigma(a_1), x_{a_1}), \cdots, (\sigma(a_n), x_{a_n}), (\sigma(b_1), x_{b_1}), \cdots, (\sigma(b_n), x_{b_n}), (\sigma(c_1), x_{c_1}), \cdots, (\sigma(c_n), x_{c_n})) \,.$$

Such a permutation implies the consistencies among circuit wires.

Example 4.2.3. *Recall the circuit in Example 4.2.1. We follow the example in the Orochi Network documentation of Copy Constraints*[11] *to derive the following copy constraints*

$$\begin{cases} x = x_{a_1} = x_{b_1} = x_{a_3} \\ y = x_{a_2} = x_{b_2} = x_{b_3} \\ z_1 = x_{c_1} = x_{a_4} \\ z_2 = x_{c_2} = x_{b_5} \\ z_3 = x_{c_3} = x_{b_4} \\ z_4 = x_{c_4} = x_{a_5} \\ z_5 = x_{c_5} = x_{a_6} \\ z_6 = x_{c_6}. \end{cases}$$

We then achieve a following partition

$$\{\{a_1, b_1, a_3\}, \{a_2, b_2, b_3\}, \{c_1, a_4\}, \{c_2, b_5\}, \{c_3, b_4\}, \{c_4, a_5\}, \{c_5, a_6\}.\{c_6\}\}.$$

We can achieve such a permutation that ensures the consistency of wires in the circuit as follows

$$\begin{aligned} &\sigma(a_1) = b_1, \quad &&\sigma(b_1) = a_3, \quad &&\sigma(a_3) = b_1, \\ &\sigma(a_2) = b_2, &&\sigma(b_2) = b_3, &&\sigma(b_3) = a_2, \\ &\sigma(c_1) = a_4, &&\sigma(a_4) = c_1, \\ &\sigma(c_2) = b_5, &&\sigma(b_5) = c_2, \\ &\sigma(c_4) = a_5, &&\sigma(a_5) = c_4, \\ &\sigma(c_5) = a_6, &&\sigma(a_6) = c_5, \\ &\sigma(c_6) = c_6. \end{aligned}$$

4.2.5 PLONKish Arithmetization

Recall in Section 4.1, PLONKish are variants that are based on PLONK arithmetization. Some additional features of PLONKish include:

- *Custom gates*: we can now add custom functions to the gate equation. The new equation

[11]https://docs.orochi.network/plonk/arithmetization/copy-constraints.html

form for a gate is

$$(q_L)x_a + (q_R)x_b + (q_O)x_c + (q_M)x_a x_b + (q_C)C + (q_F)f(x_a, x_b, x_c) = 0$$

with q_F being the new selector for the gate and f being the function of 3 variables.

- *Large fan-in and fan-out*: the PLONK arithmetization can now be extended to support more than 2 inputs and 1 output for each gate.

- *Lookup tables*: the gate equation can also be extended to allow checking that some input value is a member of a table of values. An example of this variant is PLOOKUP [GW20a].

Many variants of PLONK have been proposed since the introduction of PLONK. *TurboPLONK*[12] introduced PLONKish arithmetization with custom gates and larger fan-in and fan-out. *PlonkUp* [PFM+22] introduces PLONKish arithmetization with custom gates, larger fan-in and fan-out and lookup tables using PLOOKUP [GW20b]. Finally, the PLONKish arithmetization used in the Halo2 proof system has custom gates, larger fan-in and fan-out, and lookup tables using its own lookup arguments.

The Halo2 documentation[13] describes the PLONKish circuits in terms of a rectangular matrix of values. Hence, they refer to **rows**, **columns**, and **cells** (entries) of this matrix with the conventional meanings.

A PLONKish circuit needs a **configuration** as follows:

- A finite field $\mathbb{F}$. The cells of the matrix will be elements of $\mathbb{F}$.

- The number of columns in the matrix, and labels of these columns as being:
  - **Fixed**: Fixed values in the circuit.
  - **Advice**: The witness values.
  - **Instance**: Public inputs (shared between the prover and the verifier).

- A subset of the columns that can participate in copy constraints.

- A **maximum constraint degree**.

[12]https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf

[13]https://zcash.github.io/halo2/concepts/arithmetization.html

- A sequence of **polynomial constraints**. These are multivariate polynomials over $\mathbb{F}$ that must be evaluated to 0 for each row. The variables in a polynomial constraint may refer to a cell in a given column of the current row or a given column of another row relative to this one. The maximum degree of each polynomial is given by the maximum constraint degree. A sequence of **lookup arguments** defined over tuples of input expressions (which are multivariate polynomials as above) and **table columns**.

- The number of rows n in the matrix, n must correspond to the size of a multiplicative subgroup of $\mathbb{F}^\times$, commonly a power of 2.

- A sequence of **copy constraints**.

- The values of the fixed column in each row.

Example 4.2.4. *Recall the circuit in Example 4.2.1. We will establish said PLONKish rectangular matrix for this circuit. The main idea is to specify the matrix to contain all variables that appeared during the arithmetic circuit computation. Then, we set up gate constraints for each row and copy constraints between rows in the table. We follow the notation from the Orochi Network documentation of Halo2*[14] *to denote each row $i \in \{1, \cdots, 6\}$ in the table by*

$$(x_{a_i}, x_{b_i}, x_{c_i}, c_i, s_{add_i}, s_{mul_i}, s_{addc_i}, s_{mulc_i})$$

corresponding to the column tuple

$$(\mathsf{advice}_0, \mathsf{advice}_1, \mathsf{advice}_2, \mathsf{constant}, \mathsf{selector}_{\mathsf{add}}, \mathsf{selector}_{\mathsf{mul}}, \mathsf{selector}_{\mathsf{addc}}, \mathsf{selector}_{\mathsf{mulc}}).$$

For each row $i \in \{1, \cdots, 6\}$, we set up 4 constraints as follows:

- $s_{add_i}(x_{a_i} + x_{b_i} - x_{c_i}) = 0.$

- $s_{mul_i}(x_{a_i} x_{b_i} - x_{c_i}) = 0.$

- $s_{addc_i}(x_{a_i} + c_i - x_{c_i}) = 0.$

- $s_{mullc_i}(x_{a_i} c_i - x_{c_i}) = 0.$

[14] https://docs.orochi.network/halo2-for-dummies/plonkish/transforming-to-plonkish-arithmetization.html

This setup means that at each row, we can set the selector vector to turn on or off these gate constraints corresponding to the gates in the circuit. To put it all together, assume that we want to prove that we know 2 values x, y such that $f(x, y) = 44$. We will build a circuit $\mathcal{C}(x, y)$ such that $\mathcal{C}$ evaluates to 1 if $f(x, y) = 44$. Only we know that $x = 3, y = 4$. Table 4.2 describes the PLONKish circuit represented by columns, rows, and cells in each row for $\mathcal{C}$.

Table 4.2: PLONKish matrix for the circuit in Example 4.2.1.

Row	a_0	a_1	a_2	c	$\mathsf{s}_{\mathsf{add}}$	$\mathsf{s}_{\mathsf{mul}}$	$\mathsf{s}_{\mathsf{addc}}$	$\mathsf{s}_{\mathsf{mulc}}$
0	3	3	9	0	0	1	0	0
1	4	4	16	0	0	1	0	0
2	3	4	12	0	0	1	0	0
3	9	12	21	0	1	0	0	0
4	21	16	37	0	1	0	0	0
5	37	0	$f(x, y) = 44$	7	0	0	1	0

4.2.6 Handling Constraints in Halo2

Recall in Section 4.2.5, using PLONKish arithmetization for the desired circuit, we (as the prover) achieve a table of cell assignments that it claims satisfy the constraint system. In this section, we show how to transform this table into a polynomial expression that is true when the constraint system in the circuit is valid. Most of the specifications are followed by the Halo2 documentation of Permutation argument[15].

Handling copy constraints. From the set of copy constraints from PLONKish arithmetization in Section 4.2.5, we derive a partition of wires in equality constraints. Then we have to construct a permutation in which each subset of variables that are in a equality-constraint set form a cycle. For example, suppose we have the following copy constraints

$$\begin{cases} a = b \\ b = d \\ c = e \\ e = f. \end{cases}$$

We derive a partition $\{\{a, b, d\}, \{c, e, f\}\}$. We want to establish a permutation that maps $[a, b, c, d, e, f]$ to $[b, d, a, e, f, c]$. Finally, we need to transform this permutation into a

[15]https://zcash.github.io/halo2/design/proving-system/permutation.html

polynomial (called **permutation argument**) that is true when such a permutation exists.

Suppose we need to check a permutation of cells in m columns, represented in Lagrange basis by polynomials $v_0, \cdots, v_{m-1}$. To label each cell in m columns, set ω be the 2^k root of unity ($k \in \mathbb{N}\backslash\{0\}$) and δ be the T root of unity where $T \cdot 2^S + 1 = p$ with T odd and $S \geq k$. We use $\delta^i \omega^j \in \mathbb{F}^\times$ as the label of the cell in row j and column i of the permutation argument. We represent the permutation σ by a vector of m length, with elements are polynomials $s_i(X)$ satisfying $s_i(\omega^j) = \delta^{i'}\omega^{j'}$. We now can convert each cell (with the label by $\delta^i\omega^j$ and value by $v_i(\omega^j)$) as follows

$$cell = label \cdot \beta + value + \gamma$$

with two challenges: β helps compress the cell to a field element, and γ adds a level of randomization to the compressed result. Hence, given a permutation represented by $s_0, \cdots, s_{m-1}$ over columns represented by $v_0, \cdots, v_m$, we aim to ensure that

$$\prod_{i=0}^{m-1}\prod_{j=0}^{n-1}\left(\frac{v_i(\omega^j) + \beta \cdot \delta^i \cdot \omega^j + \gamma}{v_i(\omega^j) + \beta \cdot s_i(\omega^j) + \gamma}\right) = 1.$$

Note that $v_i(\omega^j) + \beta \cdot \delta^i \cdot \omega^j + \gamma$ represents the unpermuted cell in compression and $v_i(\omega^j) + \beta \cdot s_i(\omega^j) + \gamma$ represents the permuted cell in compression.

Let Z_P be the polynomial such that $Z_P(\omega^0) = Z_P(\omega^n) = 1$. Then for $j \in [0, n)$

$$\begin{aligned} Z_P(\omega^{j+1}) &= \prod_{h=0}^{j}\prod_{i=0}^{m-1} \tfrac{v_i(\omega^h)+\beta\cdot\delta^i\cdot\omega^h+\gamma}{v_i(\omega^h)+\beta\cdot s_i(\omega^h)+\gamma} \\ &= Z_P(\omega^j)\prod_{i=0}^{m-1} \tfrac{v_i(\omega^j)+\beta\cdot\delta^i\cdot\omega^j+\gamma}{v_i(\omega^j)+\beta\cdot s_i(\omega^j)+\gamma}. \end{aligned}$$

Then it is sufficient to randomize two challenge values β, γ and prove that

$$Z_P(\omega X) \cdot \prod_{i=0}^{m-1}\left(v_i(X) + \beta \cdot s_i(X) + \gamma\right) - Z_P(X) \cdot \prod_{i=0}^{m-1}\left(v_i(X) + \beta \cdot \delta^i \cdot X + \gamma\right) = 0$$
$$\ell_0 \cdot (1 - Z_P(X)) = 0.$$

where $l_0 = 1$ if $X = \omega^n$ and 0 if otherwise.

Handling gate constraints. To handle gate constraints in the circuit, we use the concept of the **vanishing polynomial**, defined as follows

Definition 4.2.1 (Vanishing polynomials)**.** A **vanishing polynomial** over a set $\mathbb{K}$ has a single root at each element $k \in \mathbb{K}$ and can be expressed as

$$V(X) = \prod_{k \in \mathbb{K}} (X - k).$$

If elements of $\mathbb{K}$ are n-th roots of unity over a finite field F, then we can express the vanishing polynomial as

$$V(X) = X^n - 1.$$

Example 4.2.5. *Consider the finite field $\mathbb{F}_7$ and the set $\mathbb{K} = \{1, 2, 4\}$ which are 3-rd roots of unity over $\mathbb{F}_7$. By Definition 4.2.1, we can express the vanishing polynomial over $\mathbb{K}$ as follows*

$$\begin{aligned}
V(X) &\equiv (X-1)(X-2)(X-4) && \pmod 7 \\
&\equiv (X^2 - 3X + 2)(X - 4) && \pmod 7 \\
&\equiv X^3 - 4X^2 - 3X^2 + 12X + 2X - 8 && \pmod 7 \\
&\equiv X^3 - 7X^2 + 14X - 8 && \pmod 7 \\
&\equiv X^3 - 1. && \pmod 7
\end{aligned}$$

To handle gate constraints, we need to convert each column of the table into a polynomial using Lagrange interpolation technique, combining each cell value of a column with an index ω^i for $i \in \{0, n\}$ where n is the number of rows in the table and ω is the n-th root of unity over a field $\mathbb{F}$, then we can evaluate to the i-th row gate value at index ω^i as follows

$$gate_i(X) = q_L(X)A(X) + q_R(X)B(X) + q_O(X)O(X) + q_M(X)A(X)B(X) + q_C(X)C(X) = 0.$$

A relation is valid if its polynomial is equal to zero. To show this, we form a vanishing polynomial $V(X) = X^n - 1$ over the set $\mathbb{K}$ of n-th roots of unity in the finite field $\mathbb{F}$. Then, we divide each polynomial relation by $V(X)$. If it is perfectly divisible by $V(X)$, then it equals zero over $\mathbb{K}$ as desired. This construction requires the prover to commit one polynomial per time. We can improve that by committing to all polynomial relations simultaneously as follows:

- The verifier samples a random value α and send α to the prover.

- The prover construct the quotient polynomial as follows

$$h(X) = \frac{gate_0(X) + y \cdot gate_1(X) + \cdots + y^i \cdots gate_i(X) + \cdots}{V(X)}$$

  and commit $h(X)$. The rest are followed by a standard polynomial commitment scheme.

4.3 Commitment Schemes in Halo2

Recall in Section 4.2.6 that all circuit constraints (gate constraints and copy constraints) that result from PLONKish arithmetization get transformed into polynomial constraints over a finite field $\mathbb{F}$, which evaluates to 0 if the constraint system is valid. To use these polynomial constraints in a proof system, Halo2 uses commitment schemes (specifically, polynomial commitment schemes) to polynomial constraints. Both Halo2 and its former version, Halo, use the *inner product argument* for their polynomial commitment scheme, which supports *Pedersen vector commitment* [Ped92] for the Commit phase. The **inner product argument** is the following construction: given the commitments of two vectors $\overrightarrow{a} = (a_0, a_1, \cdots, a_{n-1})$ and $\overrightarrow{b} = (b_0, b_1, \cdots, b_{n-1})$ of size n and with entries in a finite field $\mathbb{F}$, prove that their inner product $\langle \overrightarrow{a}, \overrightarrow{b} \rangle = \sum_{i=0}^{n-1} a_i b_i$ is equal to z. Their approach achieves $\mathcal{O}(n \log n)$ communication complexity in comparison to the naive solution (the prover simply sends the polynomial's coefficients to the verifier), which achieves $\mathcal{O}(n)$ communication.

Later, the Privacy & Scaling Explorations team (PSE) forked the original Halo2 project of Zcash[16] and added a new polynomial commitment scheme called the KZG polynomial commitment scheme [KZG10]. We will explain in detail the KZG commitment scheme through our proposed commitment scheme for the generalized memory in Section 5.3.3.

[16] https://github.com/privacy-scaling-explorations/halo2

Chapter 5

Generalization of the Memory

Recall that traditional approaches to zero-knowledge proofs represent the statements as arithmetic circuits, which are problematic in dealing with statements of program execution, as described in Chapter 1. Therefore, we need to convert these statements into a different model. Specifically, newer research constructs zero-knowledge proof systems in the RAM computation model. The problem statement is now proving a program in a RAM model executes correctly (which is equivalent to proving a statement is true) without disclosing any activities in the internal structure (which is equivalent to not disclosing any additional information), hence the inevitable need for zero-knowledge proofs for memory execution. It is necessary to have a memory specification in a RAM program derived from the original definition of RAM program [FKL$^+$21, dSGOTV22]. Various memory specifications have been studied and proposed by PSE[1], Risc Zero[2], and Cairo [GPR21]. In this section, the student proposes to use a solution of implementing a RAM machine module from Orochi Network[3] project zkMemory[4], which is also their trademark, that can execute some basic instructions and output the execution trace. The student also gives an approach to applying commitment schemes to commit the execution trace to use these values to prove the consistency of the memory. Note that our scope in this project does not consider some specific problems related to real memory management, such as memory paging, concurrency, etc. The "memory management" module that we follow simply means to manage simple operations to the

[1]https://pse.dev

[2]https://dev.risczero.com/api/zkvm/

[3]This capstone project was made during the student's internship at Orochi Network from 12 June 2023 to 11 June 2024, under the core supervisors such as Mr. Tran Anh Dung and Mr, Pham Nhat Minh. The student also received financial support from Orochi Network.

[4]https://github.com/orochi-network/orochimaru/tree/main/zkmemory

memory such as $\mathsf{READ}, \mathsf{WRITE}, \mathsf{PUSH}$, and POP and proposing a commitment scheme for the execution trace (see Section 5.3.1) of a RAM program.

5.1 Memory Layout

Before we begin to explain the generalized memory, we want to mention an original memory based on the definition of RAM program in Section 2.5. In the original definition of a RAM program, memory consists of only memory cells with a fixed size. Note that we aim to build a module that takes a RAM program as input, but we do not consider the translation from the original program (written in other languages or a smart contract, etc.). Therefore, if we use the original memory layout, users will find it tedious to manually translate to our RAM program. To address this issue, we need to specify the memory clearer (ideally capture the actual layout of RAM in a general computer), which means that we can define the components of the memory or define more basic instructions for the memory to keep the translated RAM program more readable and easier to translate.

In this section, the student explains a memory generalization in the definition of RAM programs in Section 2.5. We can define memory as an array of N **memory cells**. Each memory cell has a size of a constant, configurable **word size** W in bytes. Word size varies among different machines: Computer systems and CPUs that follow the x86-64 (also called AMD64, Intel64) architecture whose word size is 8 bytes (64 bits); WebAssembly is a binary instruction format for a stack-based virtual machine whose word size is 8 or 4 bytes (64 bits or 32 bits); Ethereum Virtual Machine (or EVM) is a crucial component in Ethereum Blockchain platforms which serves as a runtime environment for executing smart contracts and decentralized applications (also called dApps), EVM has the word size of 32 bytes (256 bits). The word size is configurable so that the module can be applied to a variety of available machines.

Figure 5.1 demonstrates an example of a memory with the word size of 4 bytes (32 bits). As we can see in Figure 5.1, each memory cell has 4 continuous bytes. Each byte has an address that increases by 1 for every subsequent byte. For example, in Figure 5.1, the address of the second memory cell $0x3d206a6f$ is $0x04$, which starts at the fifth byte of the memory. Each memory cell has an address that is a multiple of the word size. Also, the number of memory cells N is normally a power of 2 ($N = 2^k, k \in \mathbb{N}$).

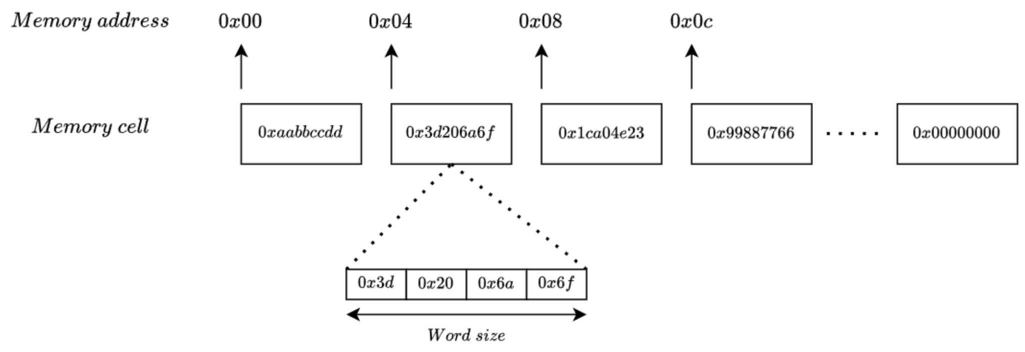


Figure 5.1: *An example of a memory with a word size of* 4 *bytes.*

From the definition of RAM program in Section 2.5, the student can modify the memory M to have more components. In a RAM computational model, there are *registers*, which hold values loaded from the memory storage to perform computations. There is also a *stack* that stores values in the Last-In-First-Out (LIFO) discipline, which serves the execution of a function or performing recursions. Some parts of the memory are designated to be read-only or forbidden to both read and write. Depending on our configuration, the student can partition the memory M into different sections.

In this project, the student modifies the memory M to consist of the following components:

- A set of **registers** $R = \{R_0, \ldots, R_{N_R}\}$ of size N_R.
- A **stack** S of size N_S (also called the **stack depth**), which is a memory with a pointer.
- 2 buffer sections B_0 and B_1, each of size N_B that represent read-only memory or forbidden parts of the memory. Buffers can also prevent out-of-bound accesses to registers, the stack, or the memory storage.
- A memory section M_S for storage of size N_{MS}.

From this modification, the student derives a layout of the memory M, which is shown in Figure 5.2.

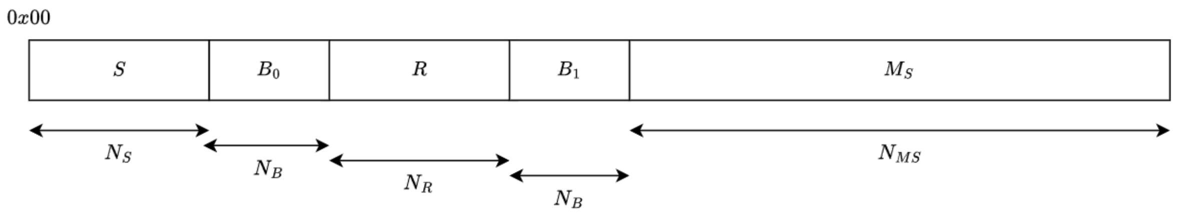


Figure 5.2: *A proposed memory layout for the RAM program.*

Note that the stack has the address of $0x00$ and 2 buffer sections B_0, B_1 are placed between main sections to prevent out-of-bound accesses. Also, it holds that

$$N_S + 2N_B + N_R + N_{MS} = N.$$

Recall that the memory used for the RAM program in Definition 2.5 consists of solely cells, with only two instructions considered, namely WRITE and READ. Using the memory layout described above provides more flexibility, as the components can be configured according to users' needs. Additionally, including stack operations introduces more practical instruction types, such as those needed for function programming and recursion. This enhancement helps users develop specific instruction sets more easily for their application projects. However, it also introduces additional conditions to consider, requiring more constraints to prove the consistency of the memory. We will explain these constraints in detail in Chapter 6.

5.2 Basic Instructions

This section defines the two most basic instructions to the memory, which are READ and WRITE. From these instructions, the student introduces two derivations that apply to the memory stack, which are PUSH and POP.

5.2.1 READ Instructions

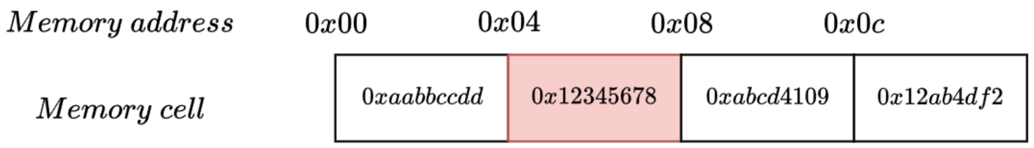
The READ instruction reads a memory cell on an address of the memory cell and assigns the result to d (which is always the last-read element) defined in a RAM program. Naturally, we want to not just read on word-multiple addresses (i.e., addresses that are multiples of the word size, which is the size of a memory cell); we also want the machine to read on the memory on non-word addresses. So, we slightly modify the READ instruction to read a word-size byte chunk starting from the address. Specifically, we define the READ instruction format as READ(x), where x is the memory's address.

There are some restrictions to the READ instruction that we must follow:

- The READ instruction must return a byte chunk of all zeros for reading the memory cell that was not written before. We assume that the memory cells are initialized by 0.
- Every READ access from the same address must equal the previous WRITE access.

The student now illustrates examples of reading on word-multiple addresses and reading on non-word addresses as follows:

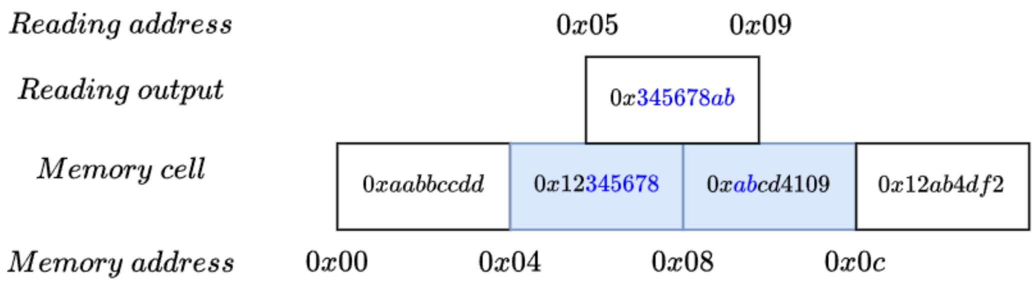
Example 5.2.1 (Read on word-multiple addresses)**.** *Figure 5.3 illustrates the* READ *instruction on a word-multiple address. We read at the address* $0x04$*, which is the address of the second memory cell, and we read all* 4 *bytes starting from address* $0x04$ *to* $0x08$*, which reads one whole memory cell. After reading,* d *is assigned to* $0x12345678$*.*



$\mathsf{READ}(0x04)$

Figure 5.3: *Read on one cell.*

Example 5.2.2 (Read on non-word addresses)**.** *Figure 5.4 illustrates the* READ *instruction on a non-word address. We read at address* $0x05$*, which is to read from the second byte of the second memory cell to the first byte of the third memory cell. Thus, the result is* $0x345678ab$*. Note that any* READ *instruction on non-word addresses can be translated to a series of basic* READ *instructions on word addresses. In this example,* $\mathsf{READ}(0x05)$ *is translated to two instructions:* $\mathsf{READ}(0x04)$ *and* $\mathsf{READ}(0x08)$*. The reading result* $0x345678ab$ *is used for later computation in the execution. However,* d *is assigned to be the memory cell at address* $0x08$*, which is* $0xabcd4109$ *because it is the last-read memory cell.*



$\mathsf{READ}(0x05) \Longrightarrow \mathsf{READ}(0x04); \mathsf{READ}(0x08)$

Figure 5.4: *Read on two cells.*

5.2.2 WRITE Instructions

In a similar construction to the READ instruction. The student defines the WRITE instruction as WRITE(x, v) where x is the memory address and v is the byte chunk of size W(bytes).

One restriction to the WRITE instruction is that we must write to the *writable* memory address (the buffer sections in the memory represent read-only or forbidden parts of the memory).

The student now gives examples of writing on word-multiple addresses and writing on non-word addresses as follows:

Example 5.2.3 (Write on word-multiple addresses)**.** *Figure 5.5 illustrates the* WRITE *instruction on a word-multiple address. We write at the address* $0x04$*, which is the address of the second memory cell, and we write* 4 *bytes starting from address* $0x04$ *to* $0x08$*. After writing,* d *is assigned to be the write chunk* $0x19193434$*.*

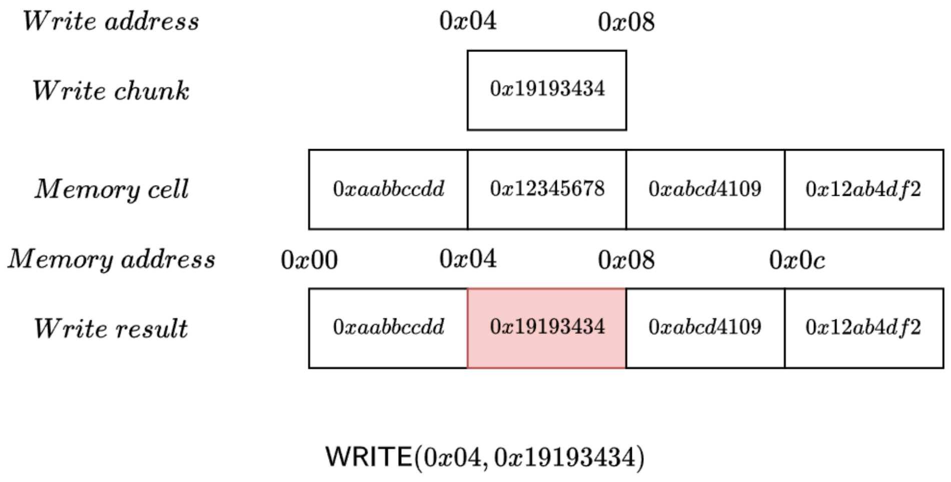
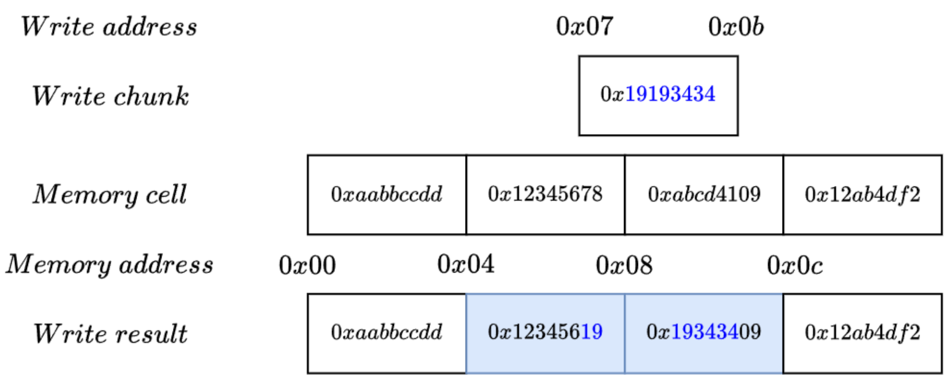


Figure 5.5: *Write on one cell.*

Example 5.2.4 (Write on non-word addresses)**.** *Figure 5.6 illustrates the* WRITE *instruction on a non-word address. We write from address* $0x07$ *to* $0x0b$*, which writes the last byte of the second memory cell and the first* 3 *bytes of the third memory cell. The* WRITE *instruction is therefore translated to* WRITE$(0x07, 0x19193434) \implies$ WRITE$(0x04, 0x00000019)$; WRITE$(0x08, 0x19343400)$*. Note that the last instruction executed is* WRITE$(0x08, 0x19343400)$*, so* d *is assigned to* $0x19343400$*. Also, we do not write the whole third cell to the value* $0x19343400$*. We just write the first* 3 *bytes, and the* *last bytes* $0x00$ *means that we do not write those bytes to the memory cell. In the example, the value after the write of the third memory cell is* $0x19343409$*, not* $0x19343400$.



$$\mathsf{WRITE}(0x07, 0x19193434) \Longrightarrow \mathsf{WRITE}(0x04, 0x00000019); \mathsf{WRITE}(0x08, 0x19343400)$$

Figure 5.6: *Write on two cells.*

5.2.3 PUSH and POP Instructions

PUSH and POP instructions are essentially derived from WRITE and READ instructions respectively. Before the student defines PUSH and POP instructions, he defines how we simulate the memory stack. The student introduces a *stack pointer* that points to a memory cell. In the memory stack, we can only read and write the memory cell that the stack pointer points to. Figure 5.7 illustrates the stack pointer in stack implementation.

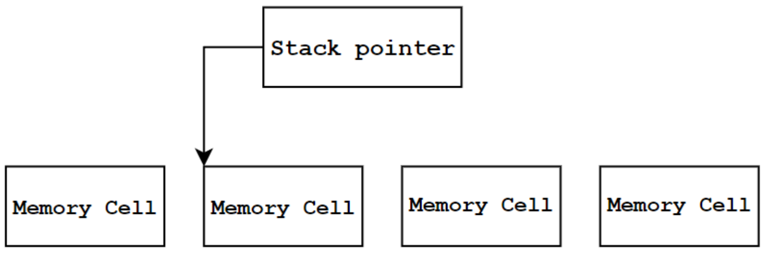


Figure 5.7: *Simulate the memory stack.*

The student defines PUSH and POP instructions as follows:

- PUSH(v): Write the byte chunk v to the memory cell pointed by the stack pointer, assign $d = v$, and then increment the stack pointer to the next memory cell. Formally:

$$\mathsf{PUSH}(v) \longrightarrow \mathsf{WRITE}(x, v)$$

where x is the address of the memory cell pointed by the stack pointer.

- $\mathsf{POP}()$: Decrement the stack pointer to the previous memory cell, read the value from the memory cell pointed by the stack pointer, which results in v, and then assign $d = v$. Formally:
$$\mathsf{POP}() \longrightarrow \mathsf{READ}(x)$$
where x is the memory cell address pointed by the stack pointer.

5.3 Execution Trace and Commitment Schemes

In this section, we define the format of an execution trace and apply several existing commitment schemes to commit the execution trace, which we can use those values after opening to prove the consistency of the memory. Note that in many projects researching zkVMs, execution trace is a common terminology.

5.3.1 Execution Trace

To prove the consistency of the memory, every memory access must be recorded. We also need to commit these records to ensure that a prover can not cheat in a private computation. After committing and revealing these records, we use them to prove the consistency of the memory.

We define the **execution trace** to be a list consists of 5-tuples defined as follows

$$(op, t, \ell, d, sd).$$

- The value op is the instruction that accesses the memory, which can either be READ, WRITE, PUSH, or POP.
- The *time log* t is an incremental value. t increases by 1 after each memory access.
- The value ℓ is the word address since we eventually translate all instructions to READ and WRITE instructions on a word address.
- The value d is the value of of the memory cell at address ℓ of size W.
- The value sd is the current depth of the memory stack (start at 0).

Example 5.3.1 (Execution table)**.** *Consider the RAM program that is equivalent to a sequence of memory accesses given that* $N = 2^{64}, W = 4$*, and the address of the stack pointer is* $0xbbbb0000$ *as follows:*

$\mathsf{WRITE}(0x00000004, 0x12345678)$

$\mathsf{PUSH}(0x12345678)$

$\mathsf{POP}()$

$\mathsf{WRITE}(0x00000009, 0xaabbccdd)$

$\mathsf{READ}(0x00000008)$.

Instead of representing the execution trace as a list, we represent it as a table, or we also call it the execution table.

Table 5.1: An execution table for the RAM program

Instruction (op)	**Time log** (t)	**Address** (ℓ)	**Value** (d)	**Stack depth** (sd)
WRITE	1	$0x00000004$	$0x12345678$	0
WRITE	2	$0xbbbb0000$	$0x12345678$	1
READ	3	$0xbbbb0000$	$0x12345678$	0
WRITE	4	$0x00000008$	$0x00aabbcc$	0
WRITE	5	$0x0000000c$	$0xdd000000$	0
WRITE	6	$0x00000008$	$0x00aabbcc$	0

5.3.2 A Proposed Scheme to Commit Trace Element using Merkle Tree

Recall that to prove the consistency of the memory, every memory access must be tracked and committed in order to prevent the prover from cheating in its private computation. Therefore, a commitment scheme for the execution trace is needed to ensure the integrity of the data and to prove the memory consistency problem. Merkle Tree is widely used in many blockchain systems such as Zcash[5], Mina Protocol[6], including Orochi Network[7] also. Therefore, this project also proposes to use Merkle Tree as one commitment scheme for the execution trace of a RAM program.

[5] https://z.cash

[6] https://minaprotocol.com/new-home

[7] https://www.orochi.network

Recall in Section 2.3, Merkle Tree is a binary tree where all the leaf nodes are the output hashes of their data. We defined the Merkle Tree in Section 2.3. This section shows how to apply Merkle Tree to commit the execution trace and create inclusion proofs.

We denote the Merkle Tree by MT with the height h, the set of 2^h data elements that are hashed to the leaf nodes of the Merkle Tree by $\mathcal{D} = \{D_i : 1 \leq i < 2^h\}$, and the Merkle Root $N_{h,0}$ by MT_{root}.

We denote the encoding scheme of the trace elements op, t, ℓ, d, sd to hash these elements as follows:

- op is encoded to a bit:
  - 0 for READ.
  - 1 for WRITE.
  - 2 for PUSH.
  - 3 for POP.
- t, ℓ, d, sd is encoded to byte representations with the length of word size W.

We denote the nodes in the Merkle Tree by $N_{i,j}$ and the i-th layer of the Merkle Tree (i.e., the list of nodes in the i-th layer of the Merkle Tree, note that the leaf node is at layer 0 and the Merkle Root is at layer $h-1$) by MT_i of length N_i where:

- j is the index of the node in i-th layer of the Merkle Tree.
- $0 \leq i < h$.
- $0 \leq j < N_i$.

We also define the **Merkle Tree Check** that holds if

$$\begin{cases} N_{0,j} = H(D_j) \text{ for all } 0 \leq j < 2^h \text{ (leaf nodes check)} \\ N_{i,j} = H(N_{i-1,2j} \parallel N_{i-1,2j+1}) \text{ for all } 1 \leq i < h, 0 \leq j < N_i \text{ (non-leaf nodes check).} \end{cases}$$

The commitment schemes using the Merkle Tree consists of four main algorithms: Setup, CommitTrace, CreateTraceProof and VerifyTraceProof. We formally describe these algorithms as follows:

Setup(h): Initializes all necessary parameters for constructing the Merkle Tree. The algorithm proceeds as follows:

Algorithm 1 Setup(h)

```
1: Initializes the Merkle Tree $MT$ with height $h$ and the data set $\mathcal{D} = \{D_i = 0 : 0 \leq i < 2^h\}$.
2: Initializes a data counter $dc = 0$. ▷ $\mathcal{D}$ is initialized with size $2^h$, but no actual data is added, so it is true that $dc = 0$.
3: for $i \leftarrow 0, 2^h$ do
4:     $N_{0,j} = H(D_j)$                          ▷ Computes the leaf nodes of the Merkle Tree
5: for $i \leftarrow 1, h$ do
6:     for $j \leftarrow 0, N_i$ do
7:         $N_{i,j} = H(N_{i-1,2j} \parallel N_{i-1,2j+1})$    ▷ Computes the non-leaf nodes up until the root
8: return $N_{h-1,0}$                              ▷ The Merkle Root
```

CommitTrace(op, t, ℓ, d, sd): Commits to a memory accesses as a 5-tuple by hashing all 5 elements to get 1 data, commits the data to the Merkle Tree, and updates the Merkle Root. The algorithm proceeds as follows:

Algorithm 2 CommitTrace(op, t, ℓ, d, sd)

```
1: $m \leftarrow H(op \parallel t \parallel \ell \parallel d \parallel sd)$
2: $D_{dc} \leftarrow m$
3: $N_{0,dc} \leftarrow H(D_{dc})$                  ▷ Recomputes the necessary nodes
4: From $N_{0,dc}$, finds its other sibling and recomputes the hashes along the path to the Merkle Root.
5: Checks if the Merkle Tree Check holds.
6: $dc \leftarrow dc + 1$
7: return $N_{h-1,0}$                              ▷ Returns the updated Merkle Root
```

CreateTraceProof(i): Creates a Merkle proof-of-inclusion for the memory access i ($i \geq 0$), which are the combined data of the 5 elements $(op_i, t_i, \ell_i, d_i, sd_i)$ and outputs the Merkle proof π_i and the 5-tuple trace elements $(op_i, t_i, \ell_i, d_i, sd_i)$. The algorithm proceeds as follows:

Algorithm 3 CreateTraceProof(i)

```
1: Initializes the Merkle proof-of-inclusion $\pi_i$ as an empty list
2: Append $N_{0,j}$ to $\pi_i$
3: Along the path from the node $N_{0,i}$ to the Merkle Root, finds its sibling (denoted by $N'$) and appends $N'$ to $\pi_i$.
4: Appends $MT_{root}$ to $\pi_i$
5: return $((op_i, t_i, \ell_i, d_i, sd_i), \pi_i)$
```

<u>VerifyTraceProof$((op_i, t_i, \ell_i, d_i, sd_i), \pi_i)$</u>: Verifies the Merkle proof π_i for the execution trace element i, which is a 5-tuple $(op_i, t_i, \ell_i, d_i, sd_i)$. If so, the algorithm outputs 1, otherwise outputs 0. The algorithm proceeds as follows:

Algorithm 4 VerifyTraceProof$((op_i, t_i, \ell_i, d_i, sd_i), \pi_i)$

```
1: $m_i \leftarrow H(op_i \parallel t_i \parallel \ell_i \parallel d_i \parallel sd_i)$
2: if $H(m_i) \neq \pi_{i_0}$ then    ▷ Check that $m_i$ is equal to the leaf node holding the trace element
3:     return 0
4: for $j \leftarrow 1, len(\pi_i) - 1$ do    ▷ Updates the hash $m_i$ along the path to the root
5:     $m_i \leftarrow H(m_i \parallel \pi_{i_j})$
6: if $m_i = \pi_i[-1]$ then    ▷ Check that $m_i$ is equal to the Merkle Root
7:     return 1
8: else
9:     return 0
```

Complexity analysis.

The Setup algorithm initializes a Merkle Tree from the data set $\mathcal{D}$ of length $n = 2^k, k \in \mathbb{N}^*$. The total nodes in the Merkle Tree is

$$n + \frac{n}{2} + \cdots + 1 = 1 + 2 + \cdots + 2^k = 2^{k+1} - 1 = 2n - 1.$$

Therefore, the space complexity of the Setup algorithm is $\mathcal{O}(n)$. Since the CommitTrace algorithm requires recomputing the nodes from the path to the leaf node to the Merkle Root, the time complexity is $\mathcal{O}(\log_2 n)$. CreateTraceProof and VerifyTraceProof both share the same time complexity of $\mathcal{O}(\log_2 n)$ since they also require computation of the nodes in the path from the leaf node to the Merkle Root. The Merkle proof-of-inclusion also includes all the nodes in the path from the leaf node to the root, which means that the Merkle proof size complexity is $\mathcal{O}(\log_2 n)$. We summarize the complexity of the commitment scheme using Merkle Tree in table 5.2.

Table 5.2: Complexity of the commitment scheme using the Merkle Tree.

	Construction	Commit	Create Proof	Proof Size	Verify
Complexity	$\mathcal{O}(n)$	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(\log_2 n)$

5.3.3 KZG Polynomial Commitment Scheme for Committing Trace Records

The KZG Polynomial commitment scheme [KZG10] performs on the data represented as polynomials. Due to its advantages, such as constant proof sizes, even in multi-proofs, and constant verification time from the verifier, KZG is a more efficient commitment scheme than the Merkle Tree. Still, it is also harder to implement related cryptographic primitives such as elliptic curves, bilinear pairings, etc. PSE[8] implemented a utility library[9] for implementing the KZG commitment scheme which is widely used. Orochi Network first proposed to use the KZG Polynomial commitment scheme to the execution trace[10]. Therefore, we also propose to use KZG as another commitment scheme for the execution trace.

In the construction of $\mathsf{PolyCommit}_{\mathsf{DL}}$ [KZG10], they used type 1 bilinear pairings over elliptic curve groups. We apply KZG with the exception that we use pairings on different groups of the same prime order p.

Before defining the algorithms, we explain how to commit to a trace element. In polynomial commitment schemes, we commit polynomials, so we need a method of converting the trace element into a polynomial. Firstly, we convert each element in the 5-tuple trace element into elements in $\mathbb{F}_p$ where p is the prime order of the group $\mathbb{G}$. Note that op is first encoded as described in Section 5.3.1. Secondly, we attach each said $\mathbb{F}_p$ element to successive powers of the first primitive root of $\mathbb{F}_{|}$, namely, $\omega, \omega^2, \omega^3$, ω^4, and ω^5. Specifically, we want to have the polynomial of degree seven, which is uniquely formed from eight points, a power of two. By doing this, the Lagrange interpolation time complexity can be reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ where n is the degree of the Lagrange interpolation polynomial. Also, setting the

[8]https://pse.dev

[9]https://github.com/privacy-scaling-explorations/halo2/tree/kzg-back/src/poly/multiopen

[10]https://hackmd.io/@chiro-hiro/SkqNGtcW2

polynomial degree to be the power of two is compatible with available implementations such as Halo2-PSE polynomial implementation[11]. Formally, we define an algorithm $\mathsf{TraceToPoly}$ which takes (op, t, l, d) as inputs and outputs the polynomial $\phi(x)$ of degree seven that is interpolated from eight points

$$(\omega, op), (\omega^2, t), (\omega^3, \ell), (\omega^4, d), (\omega^5, 0), (\omega^6, 0), (\omega^7, 0), (\omega^8, 0)$$

, where ω is the first primitive root of the group $\mathbb{G}$.

We now apply the KZG Polynomial Commitment Scheme to consist of 5 algorithms: Setup, $\mathsf{CommitTrace}$, $\mathsf{VerifyTrace}$, $\mathsf{CreateTraceWitness}$, and $\mathsf{VerifyTraceWitness}$, described as follows:

$\underline{\mathsf{Setup}(1^\lambda, t = 8)}$: Initializes necessary parameters for the commitment scheme. The algorithm proceeds as follows:

- Computes three groups $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ of prime order p providing the λ-bit security parameter, such that there exist a bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ and for which the t-SDH assumption holds.
- Denotes the generated bilinear groups as $\mathcal{G} = (e, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$.
- Samples a random generator points $G_1 \xleftarrow{\$} \mathbb{G}_1$ and $G_2 \xleftarrow{\$} \mathbb{G}_2$, and a secret key $\mathsf{sk} \xleftarrow{\$} \mathbb{F}_p$.
- Computes two **common reference strings** (or crs) crs_1 on group $\mathbb{G}_1$ and crs_2 on group $\mathbb{G}_2$ by setting $\mathsf{crs}_1 = (G_1, \mathsf{sk}G_1, \ldots, \mathsf{sk}^t G_1)$ and setting $\mathsf{crs}_2 = (G_2, \mathsf{sk}G_2, \ldots, \mathsf{sk}^t G_2)$.
- Outputs the public key $\mathsf{pk} = (\mathcal{G}, \mathsf{crs}_1, \mathsf{crs}_2)$ and delete sk. Note that it is crucial to delete sk because sk is not needed for the rest of the algorithms in our scheme, and if some adversary successfully retrieves the undeleted sk then the commitment scheme is broken.

Provided that we found such three groups $\mathbb{G}_1, \mathbb{G}_2$, and $\mathbb{G}_T$ of prime order p, we can represent the algorithm using the following pseudocode as described below

[11] https://github.com/privacy-scaling-explorations/halo2/blob/main/halo2_proofs/src/poly/kzg/commitment.rs

Algorithm 5 $\mathsf{Setup}(1^\lambda, \mathsf{t} = 8)$

```
1: $G_1 \overset{\$}{\leftarrow} \mathbb{G}_1$    ▷ Samples a generator point $G_1$ in $\mathbb{G}_1$
2: $G_2 \overset{\$}{\leftarrow} \mathbb{G}_2$    ▷ Samples a generator point in $\mathbb{G}_2$
3: $\mathsf{sk} \overset{\$}{\leftarrow} \mathbb{F}_p$    ▷ Samples a secret key in $\mathbb{F}_p$
4: $\mathsf{crs}_1 \leftarrow (G_1, \mathsf{sk}G_1, \dots, \mathsf{sk}^{\mathsf{t}}G_1)$
5: $\mathsf{crs}_2 \leftarrow (G_2, \mathsf{sk}G_2, \dots, \mathsf{sk}^{\mathsf{t}}G_2)$
6: Delete $\mathsf{sk}$    ▷ This is important for security issues
7: return $\mathsf{pk} = (\mathsf{crs}_1, \mathsf{crs}_2)$.
```

$\underline{\mathsf{CommitTrace}(\mathsf{pk}, (op, t, \ell, d, sd))}$: Commits to a trace element and outputs the commitment $\mathcal{C}$ for the polynomial representing the trace element. The algorithm proceeds as follows:

- Computes the polynomial representing the trace element $\phi(x) = \mathsf{TraceToPoly}(op, t, \ell, d, sd)$.
- Asserts that $deg(\phi) \leq \mathsf{t}$. If not, the algorithm terminates with an error.
- Denotes $\phi(x) = \sum_{j=0}^{deg(\phi)} \phi_j x^j$.
- Computes the commitment $\mathcal{C}$ to the polynomial $\phi(x)$ using the public key pk

$$\mathcal{C} = \phi(s)G_1 = \sum_{j=0}^{deg(\phi)} \phi_j s^j G_1 = \sum_{j=0}^{deg(\phi)} \phi_j (s^j G_1) = \sum_{j=0}^{deg(\phi)} \phi_j \mathsf{crs}_{1_j}.$$

- Outputs $\mathcal{C}$.

We can represent the algorithm by the following pseudocode

Algorithm 6 $\mathsf{CommitTrace}(\mathsf{pk}, (op, t, \ell, d, sd))$

```
1: $\phi(x) \leftarrow \mathsf{TraceToPoly}(op, t, \ell, d, sd)$
2: if $deg(\phi) > \mathsf{t}$ then
3:     return Error
4: $\mathcal{C} \leftarrow \sum_{j=0}^{deg(\phi)} \phi_j \mathsf{crs}_1[j]$
5: return $\mathcal{C}$.
```

$\underline{\mathsf{VerifyTrace}(\mathsf{pk}, (op, t, \ell, d, sd), \mathcal{C})}$: Verifies that $\mathcal{C}$ is indeed the commitment to the polynomial representing the trace element (op, t, ℓ, d, sd) as follows

Algorithm 7 $\mathsf{VerifyTrace}(\mathsf{pk}, (op, t, \ell, d, sd), \mathcal{C})$

```
1: $\phi(x) \leftarrow \mathsf{TraceToPoly}(op, t, \ell, d, sd)$
2: $\mathcal{C}' \leftarrow \sum_{j=0}^{deg(\phi)} \phi_j \mathsf{crs}_1[j]$    ▷ *Recomputes the commitment*
3: **if** $\mathcal{C} = \mathcal{C}'$ **then**
4:     **return** $1$
5: **else**
6:     **return** $0$.
```

$\underline{\mathsf{CreateTraceWitness}(\mathsf{pk}, (op, t, \ell, d, sd)))}$: Creates the proof π for the polynomial representing the trace element (op, t, l, d). The algorithm proceeds as follows:

- Computes the polynomial representing the trace element $\phi(x) = \mathsf{TraceToPoly}(op, t, \ell, d, sd)$.
- Computes $L(x)$ which is the Lagrange interpolation polynomial of 5 points $(\omega, \phi(\omega)), (\omega^2, \phi(\omega^2)), (\omega^3, \phi(\omega^3)), (\omega^4, \phi(\omega^4)), (\omega^5, \phi(\omega^5))$.
- Computes $Z(x) = \prod_{i=1}^{5}(x - \omega^i)$.
- Computes $\psi(x) = \dfrac{\phi(x) - L(x)}{Z(x)}$.
- Computes $w = \psi(s)G_2 = \sum_{j=0}^{deg(\psi)} \psi_j(s^j G_2)$.
- Outputs $\pi = ((\omega, \omega^2, \omega^3, \omega^4, \omega^5), (op, t, \ell, d, sd), w)$.

We can represent the algorithm using the following pseudocode

Algorithm 8 $\mathsf{CreateTraceWitness}(\mathsf{pk}, (op, t, \ell, d, sd)))$

```
1: $\phi(x) \leftarrow \mathsf{TraceToPoly}(op, t, \ell, d, sd)$
2: $L(x) \leftarrow \mathsf{LagrangeInterpolate}\big((\omega, \phi(\omega)), (\omega^2, \phi(\omega^2)), (\omega^3, \phi(\omega^3)), (\omega^4, \phi(\omega^4)), (\omega^5, \phi(\omega^5))\big)$
3: $Z(x) \leftarrow \prod_{i=1}^{5}(x - \omega^i)$
4: $\psi(x) \leftarrow \dfrac{\phi(x) - L(x)}{Z(x)}$
5: $w \leftarrow \sum_{j=0}^{deg(\psi)} \psi_j \mathsf{crs}_2[j]$ ▷ *Essentially the same when we commit to a polynomial, with the exception that we do in $\mathbb{G}_2$*
6: $\pi \leftarrow ((\omega, \omega^2, \omega^3, \omega^4, \omega^5), (op, t, \ell, d, sd), w)$
7: **return** $\pi$.
```

$\underline{\mathsf{VerifyTraceWitness}(\mathsf{pk}, \mathcal{C}, \pi = ((\omega, \omega^2, \omega^3, \omega^4, \omega^5), (op, t, \ell, d, sd), w)}$: Verifies that π is indeed the proof for the trace element (op, t, ℓ, d, sd) as follows:

- Computes $L(x)$ which is the Lagrange interpolation polynomial of 5 points $(\omega, op), (\omega^2, t), (\omega^3, \ell), (\omega^4, d), (\omega^5, sd)$.
- Computes $Z(x) = \prod_{i=1}^{5}(x - \omega^i)$.
- Verifies that $e(\mathcal{C} - L(\mathsf{sk})G_1, G_2) = e(Z(\mathsf{sk})G_1, w)$. It outputs 1; otherwise, it outputs 0.

We can use the following pseudocode to describe the algorithm better

Algorithm 9 $\mathsf{VerifyTraceWitness}(\mathsf{pk}, \mathcal{C}, \pi = ((\omega, \omega^2, \omega^3, \omega^4, \omega^5), (op, t, \ell, d, sd), w)$

```
1: $\phi(x) \leftarrow \mathsf{TraceToPoly}(op, t, \ell, d, sd)$
2: $L(x) \leftarrow \mathsf{LagrangeInterpolate}\big((\omega, \phi(\omega)), (\omega^2, \phi(\omega^2)), (\omega^3, \phi(\omega^3)), (\omega^4, \phi(\omega^4)), (\omega^5, \phi(\omega^5))\big)$
3: $Z(x) \leftarrow \prod_{i=1}^{5}(x - \omega^i)$
4: if $e(\mathcal{C} - L(\mathsf{sk})G_1, G_2) = e(Z(\mathsf{sk})G_1, w)$ then
5:     return 1
6: else
7:     return 0.
```

Remark. The $\mathsf{VerifyTraceEval}$ algorithm correctly verifies the witness because

$$e(\mathcal{C} - L(\mathsf{sk})G_1, G_2) = e(G_1, G_2)^{\phi(\mathsf{sk}) - L(\mathsf{sk})} = e(Z(\mathsf{sk})G_1, w) = e(G_1, G_2)^{Z(\mathsf{sk})\psi(\mathsf{sk})}$$

$$\iff \psi(\mathsf{sk}) = \frac{\phi(\mathsf{sk}) - L(\mathsf{sk})}{Z(\mathsf{sk})}.$$

Complexity analysis. We will provide an analysis of the prover's complexity, the verifier's, and the scheme's proof size as follows:

- **The prover.** The prover has to commit to a polynomial of degree no less than t and provide a proof for 4 evaluations of a said polynomial. Also, since we use successive multiplicative generators in the $\mathsf{TraceToPoly}$ algorithm, the interpolation can be done recursively (also known as the *Inverse Fast Fourier Transform*), the complexity of the $\mathsf{TraceToPoly}$ algorithm is $\mathcal{O}(\mathsf{t} \log \mathsf{t})$. Therefore, the prover time complexity is $\mathcal{O}(\mathsf{t} \log \mathsf{t})$.
- **The verifier.** Since the verifier just needs to perform one bilinear pairing, the time complexity of the verifier is $\mathcal{O}(1)$.

- **The proof size**. The proof is a single element in the group $\mathbb{G}_2$, regardless of how many evaluations the prover needs to prove. Hence, the proof size complexity is also $\mathcal{O}(1)$.

Chapter 6

Proving Memory Consistency Using Halo2

In Chapter 5, we recall a method of generalizing the memory used in any RAM program by Orochi Network[1], which enables some advantages compared with the old memory layout. The memory executes programs and outputs an execution trace capturing all memory accesses during execution, which is guaranteed security and integrity using commitment schemes. Recall in Chapter 4 that to prove a statement in zero-knowledge, we need to represent the statement as a constraint or a constraint system that is compatible with its intended computation model (we may express the statement as Boolean constraints if we want to represent said statement as a Boolean circuit, arithmetic circuits behave similarly). Therefore, it is necessary to define the statement "The memory is consistent throughout the execution of a RAM program P" into a constraint system in order to express the statement as an arithmetic circuit. In this chapter, along with four constraints 6.1, 6.2, 6.3, and 6.5 that Franzese et al. [FKL+21] and Orochi Network use, we propose two additional constraints 6.6, 6.10, and modify constraint 6.4 to 6.11 that can check the memory consistency of any RAM program execution, which is followed by consistency circuits of Guilhem et al. [dSGOTV22] and Franzese et al. [FKL+21]. Recall that when we say a memory is *consistent*, we mean that if you last write to a memory address at time t with value x, then at any time $t' > t$, reading from that memory address also returns x.

[1]https://github.com/orochi-network/orochimaru/tree/main/zkmemory#readme

6.1 Constraints for proving memory consistency

In this section, we introduce the constraint system of the solution by Franzese et al. [FKL+21] to prove the memory consistency. The memory generalized by the author did not have the stack component, so the trace record of the execution trace $\mathcal{L}$ only has 4 components: the address ℓ_i, the time log t_i, the operation $op_i \in \{0; 1\}$, and the value d_i. The author sorts the execution trace $\mathcal{L}$ by address, then by time log, resulting in the updated trace called $\mathcal{L}'$. We explain the separate constraint in the circuit that checks the memory consistency of the author's solution.

Permuted execution trace sorting check. This checks that the execution trace $\mathcal{L}'$ is sorted correctly by address ℓ, then by time log t. Hence the constraint for this check is expressed as follows for $i \in \{0; \cdots ; n-2\}$

$$(\ell'_{i+1} > \ell'_i) \vee \left((\ell'_{i+1} = \ell'_i) \wedge (t'_{i+1} > t'_i)\right).$$

Each read of a memory address returns the last value written there. This check is performed by the following constraint for $i \in \{0, \cdots , n-2\}$

$$(\ell'_{i+1} \neq \ell'_i) \vee (d'_{i+1} = d'_i) \vee (op'_i = 1).$$

Suppose the opposite is true, which means

$$(\ell'_{i+1} = \ell'_i) \wedge (op'_i = 0) \wedge (d'_{i+1} \neq d'_i).$$

Then it would be a contradiction since both operations of an adjacent trace record are all READ type, but the values at the same address are different, which violates the consistency requirement.

Each memory is written before it is read. This check consider the permuted execution trace $\mathcal{L}'$ and is expressed as follows

$$(\ell'_{i+1} = \ell'_i) \vee (op'_{i+1} = 1).$$

Suppose the opposite is true, which means

$$(\ell'_{i+1} \neq \ell'_i) \wedge (op'_{i+1} = 0).$$

Since $\mathcal{L}'$ is sorted and $\ell'_{i+1} \neq \ell'_i$, the access of address ℓ'_{i+1} is the first memory access, so it should be the WRITE instruction. However, $op'_{i+1} \neq 1$, so it would be a contradiction.

Checking the first instruction. This check ensures the initial state of the memory through the first trace record of $\mathcal{L}$. Since all addresses must be written before they can be read, the first operation should be WRITE, and the time log should start at 0. Put it all together, and we have the constraint as follows

$$(op_0 = 1).$$

All addresses are within bound. This check ensures that all the address values in the execution trace are within a bound N where N is the size of the memory. In the memory defined by the author, each address value is in $[0, N-1]$. Therefore, the author achieves the constraint by ensuring the last address in $\mathcal{L}'$, called ℓ'_{k-1} where k is the size of the execution trace, is smaller than N since ℓ_{k-1} is the largest address value in $\mathcal{L}'$ (assume that $\mathcal{L}'$ is sorted correctly). Therefore, the constraint for this check is

$$\ell'_{k-1} < N.$$

Finally, we combine all the component checks of the consistency check into a constraint system as follows

$$\begin{cases} (\ell'_{i+1} > \ell'_i) \vee \big((\ell'_{i+1} = \ell'_i) \wedge (t'_{i+1} > t'_i)\big) & (6.1) \\ (\ell'_{i+1} \neq \ell'_i) \vee (d'_{i+1} = d'_i) \vee (op'_i = 1) & (6.2) \\ (\ell'_{i+1} = \ell'_i) \vee (op'_{i+1} = 1) & (6.3) \\ op'_0 = 1 & (6.4) \\ \ell'_{k-1} < N. & (6.5) \end{cases}$$

6.2 Our proposal for proving the consistency of the generalized memory

Recall that we have an execution trace resulted from a certain RAM program execution $\mathcal{L} = [(\ell_0, t_0, op_0, d_0, sd_0), \cdots, (\ell_{n-1}, t_{n-1}, op_{n-1}, d_{n-1}, sd_{n-1})]$ where n is the total number of instructions executed in the RAM program. We want to prove that this execution trace is consistent, meaning that if you last write to a memory address at time t with value x, then at any time $t' > t$, reading from that memory address also returns x.

Adapting the approach of Franzese et al. [dSGOTV22], described in Section 6.1, we also need a permuted version of the execution trace $\mathcal{L}$, namely $\mathcal{L} = [(\ell'_0, t'_0, op'_0, d'_0, sd'_0), \cdots, (\ell'_{n-1}, t'_{n-1}, op'_{n-1}, d'_{n-1}, sd'_{n-1})]$, in which all trace records in $\mathcal{L}'$ are sorted first by the address ℓ, then by value t, and finally by the operation op. In our proposal, we still encode the instruction by the following map: 0 for READ and 1 for WRITE.

We propose a set of constraints that can prove the consistency of the memory if they are all valid. We essentially retain all the constraints from the solution by Franzese et al. [FKL+21], with a slight modification that all time log values must increment by exactly 1 for each instruction execution (note that this is ensured by the solution of Guihem et al. [dSGOTV22]). In addition, we ensure that the stack depth after each instruction execution must be either unchanged, increased by 1, or decreased by 1. We now explain the separate constraints proposed to prove the memory consistency problem.

Time log sorting check. If $\mathcal{L}$ is correctly sorted by time log (chronologically sorted), then for $i \in \{0, \cdots, n-2\}$:

$$t_{i+1} - t_i = 1.$$

Permuted execution trace sorting check. This check is for the permuted version of $\mathcal{L}$, which is $\mathcal{L}'$. For $i \in \{0, \cdots, n_2\}$:

$$(\ell'_{i+1} > \ell'_i) \vee \big((\ell'_{i+1} = \ell'_i) \wedge (t'_{i+1} > t'_i)\big).$$

This constraint ensures that all elements in $\mathcal{L}$ are either sorted by the address (the rest of the sorting criteria are not necessary to prove) or sorted by the time log if they have the same value of the address.

Each read of a memory address returns the last value written there. This check is performed by the following constraint for $i \in \{0, \cdots, n_2\}$

$$(\ell'_{i+1} \neq \ell'_i) \vee (op'_i = 1) \vee (d'_{i+1} = d'_i).$$

Suppose the opposite is true, which means

$$(\ell'_{i+1} = \ell'_i) \wedge (op'_i = 0) \wedge (d'_{i+1} \neq d'_i).$$

Then it would be a contradiction since both operations of an adjacent trace record are all READ type, but the values at the same address are different, which violates the consistency requirement.

Operation value bound check. Since we only consider 4 basic instructions which encodes to the set $\{0;1\}$, our constraint will be expressed as follows for $i \in \{0;\cdots;n-1\}$

$$op'_i \in \{0;1\}.$$

Each memory is written before it is read. This check consider the permuted execution trace $\mathcal{L}'$ and is expressed as follows

$$(\ell'_{i+1} = \ell'_i) \vee (op'_{i+1} = 1).$$

Suppose the opposite is true, which means

$$(\ell'_{i+1} \neq \ell'_i) \wedge (op'_{i+1} \neq 1).$$

Since $\mathcal{L}'$ is sorted and $\ell'_{i+1} \neq \ell'_i$, the access of address ℓ'_{i+1} is the first memory access, so it should be the WRITE instruction. However, $op'_{i+1} \neq 1$, so it would be a contradiction.

Checking the first instruction. This check is the same as the check described in Section 6.1, with an addition of the time log value should start by 0. Put it all together, and we have the constraint as follows

$$(op_0 = 1) \vee (t_0 = 0).$$

Stack depth check. This check considers the original execution trace $\mathcal{L}$ and ensures that the stack depth value sd_i at any time log t_i must either be unchanged, increase by exactly 1, or decrease by exactly 1. Hence, the constraint is expressed as follows

$$sd_i - sd_{i-1} \in \{-1, 0, 1\}.$$

Finally, we combine all the component checks of the consistency check into a constraint system as follows

$$\begin{cases} (t_{i+1} - t_i = 1) & (6.6) \\ (\ell'_{i+1} > \ell'_i) \vee \big((\ell'_{i+1} = \ell'_i) \wedge (t'_{i+1} > t'_i)\big) & (6.7) \\ (\ell'_{i+1} \neq \ell'_i) \vee (op'_{i+1} = 1) \vee (d'_{i+1} = d'_i) & (6.8) \\ (\ell'_{i+1} = \ell'_i) \vee (op'_{i+1} = 1) & (6.9) \\ (sd_i - sd_{i-1}) \in \{-1, 0, 1\} & (6.10) \\ (op_0 = 1) \wedge (t_0 = 0) & (6.11) \\ \ell'_{k-1} < N. & (6.12) \end{cases}$$

Chapter 7

Implementation

In this chapter, we demonstrate an available implementation called zkMemory that simulates the memory layout and solves the memory consistency problem. zkMemory is a universal memory commitment module implemented by Orochi Network[1] that can be used for any zkVMs. The work received a grant from the Ethereum Foundation, and the author of this thesis is also one of the core contributors to the project. The complete implementation of the whole project is the property of Orochi Network and is publicly available at the link: https://github.com/orochi-network/orochimaru/tree/main/zkmemory. In this chapter, we explain some of the most important modules in the project zkMemory. Later, we conduct an evaluation by forking the project and adding some modules for evaluation. The forked project by the student can be found here: https://github.com/HappyFalcon22/orochimaru/tree/main/zkmemory.

7.1 Programming Languages and Libraries Selection

zkMemory aims to implement a module that balances the performance, memory safety, and availability of existing libraries that are useful for future works. They use the Halo2 proof system to build the memory consistency circuit based on the constraint system described in Section 6.2. Apparently, the programming language used to implement the module is **Rust programming language**. Rust employs the unique borrow checker, which prevents many potential errors, including buffer overflows and using variables after being deallocated, a feature that C and Python lack. Rust implements Halo2, which is a well-known cryptosystem

[1] https://orochi.network/

used for blockchain-based applications. The implementation of Halo2 by PSE implements KZG polynomial commitment scheme [KZG10], while PSE's *halo2curves* library implements pairing-friendly elliptic curves[2]. One big disadvantage of Rust is that the language itself is challenging to learn due to its complex mechanism of ownership and borrowing. Moreover, Rust uses structs and traits, which employ OOP in a very different view compared to other programming languages.

Specifically, the Halo2 version of the Privacy & Scaling Exploitation team[3] is used for implementing commitment schemes and proving the consistency of the memory. For dealing with big numbers, they use the crate *ethnum*[4], which implements unsigned 256-bit integers types. They also used the PSE's *halo2curves* library to implement pairing-friendly elliptic curves into the polynomial commitment scheme. However, since the project was in the early stages, only a simplified memory consistency circuit where constraints regarding the stack are removed was implemented.

7.2 Implementation of the memory

In Rust, to create utilities and flexibility for users to apply to their desired project, it is common practice to implement a Rust *trait* instead of a traditional "class". For users to apply to their own project, they need to build their own Rust *structs* using our implemented trait. zkMemory implements a trait called *AbstractMachine*, which simulates a machine with configurable word size according to the generalization of the memory in chapter 5. It can support machines with word sizes of $256, 128, 64, 32$, and 16 bits. The implemented machine can execute a sequence of instructions derived from a RAM program, push each trace record into an execution trace during each instruction execution, and additional utilities for internal programming, including outputting the execution trace, getting the maximum value of the stack depth, etc.

The components of the machine as a whole include a memory machine, a register machine, and a stack machine.

Memory machine. The trait implementing the memory machine is *AbstractMemoryMachine* trait with two basic functionalities: read the write to a memory address. They use the *Red-Black Tree* as the data structure for the memory storage since

[2]https://github.com/privacy-scaling-explorations/halo2curves

[3]https://github.com/privacy-scaling-explorations/halo2

[4]https://docs.rs/ethnum/latest/ethnum/

most of the operations performed with the storage are insertions. Red-Black Tree handles insertions well with time complexity being $\mathcal{O}(\log_2 n)$.

Register machine. They first implement a struct *Register* with a method to initialize and assign the desired location of the register in the memory since unlike stack, the total number of registers and the number of active registers really depends on each user's specification of the memory. The *AbstractRegisterMachine* trait is implemented to support 3 main methods: initializing a new register in the register region, getting the value of a register, and setting the value of a register.

Stack machine. They implement a *StackMachine* trait that can support PUSH and POP instructions and can handle exceptions during execution, such as overflowed and underflowed stack. The execution method matches the PUSH and POP execution described in the proposal in chapter 5.

To capture all memory accesses during RAM program execution, they use a struct named *TraceRecord* with possible instructions being READ and WRITE (since the current implementation solves the problem where stack constraints are removed, all instructions are currently translated to 2 basic instructions READ and WRITE).

7.3 Implementation of KZG Commitment Scheme

Next, they implement a struct *KZGCommitmentScheme*, with four main functionalities: initialize the necessary parameters for the scheme, commit a trace record (returns the commitment value), open all trace records in the execution trace, and verify all trace records in the execution trace. Since all necessary utilities needed to implement are implemented by the PSE team in their Halo2 project, they just need to reuse their curve specifications and other parameters in our implementation. The curve that they use in this implementation is the *BN256* pairing-friendly elliptic curve group[5].

Figure 7.1 illustrates an example protocol that leverages the KZG polynomial commitment scheme. The protocol involves three parties: Prover, Verifier, and a trusted third party for generating the necessary parameters for the scheme. To begin, the *Setup* function generates all parameters for the scheme, returning the public key to the prover and the verifier while destroying the private key in case of a third-party compromise. The prover can commit to a trace record after each instruction execution using the *CommitTrace* method, which returns

[5] https://github.com/privacy-scaling-explorations/halo2curves/tree/main/src/bn256

the commitment value $\mathcal{C}$ for both parties. The prover can then generate proof for the i-th trace record (i is normally demanded by the Verifier) using *CreateTraceWitness* function, while the verifier can check the proof using *VerifyTraceWitness* function. For the open witness and verify phase, zkMemory implements its more efficient version compared to the proposal described in Chapter 5, which is described in [BDFG20]. This implementation further mitigates the verifier complexity but requires more group element computation complexity.

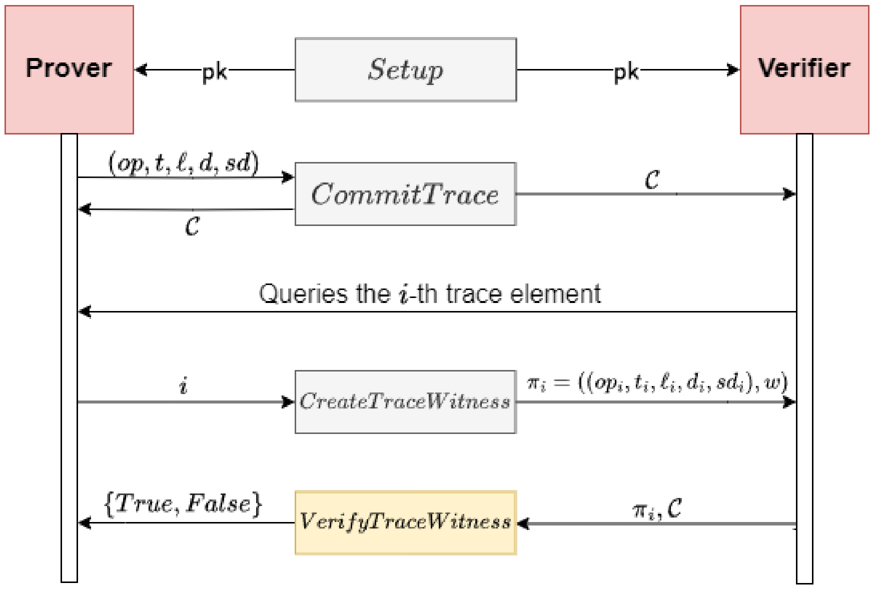


Figure 7.1: *A commitment scheme protocol based on KZG.*

7.4 Memory Consistency Circuit

The last thing to do is to build a circuit that can prove the memory consistency problem. zkMemory implements 3 submodules for handling separate constraints for permutation check (for checking if two execution traces - the original one and the sorted one - are permutations of each other), constraints for the original execution trace, and constraints for the sorted execution trace). They also implement a utility module *gadget* that implements some necessary subcircuits such as inequality circuits, lookup circuits, and IsZero circuits (to prove that a number is a zero in an arithmetic context, they set the rules that if the number is non-zero, then its product with its inverse must equal to 1).

Original execution trace sorting check. This check is implemented in module *original_memory_circuit*. The configuration in Halo2 used for this circuit includes the trace records in the execution trace, selectors, a subconfiguration for implementing the circuit that handles inequality constraints, and a lookup table for handling lookup constraints (for example, we want to build a circuit that proves a number x is at most 8 bits long - which is in the range of $[0; 2^8 - 1]$). The constraints of the original execution trace include the first time log being 0, and the trace is sorted correctly by time log.

```rust
pub(crate) struct OriginalMemoryConfig<F: Field + PrimeField> {
    /// The original trace circuit
    pub(crate) trace_record: TraceRecordWitnessTable<F>,
    /// The selectors
    pub(crate) selector: Column<Fixed>,
    pub(crate) selector_zero: Selector,
    /// The config checking the correct sorting by time log
    pub(crate) greater_than: GreaterThanConfig<F, 3>,
    /// The lookup table
    pub(crate) lookup_tables: LookUpTables,
}
```

Listing 1: Configuration for checking constraints from the original execution trace.

Sorted execution trace check. This check is implemented in the module *sorted_memory_circuit*. The configuration and custom gate handling constraints are described in the code below.

Permutation check. This check is implemented in module *permutation_circuit*, which take advantage of the **shuffle** API from PSE implementation[6] so the workload on this check is mitigated.

Memory consistency circuit. To put it all together, the code below depicts the configuration and the circuit struct for our final module *memory_consistency_circuit*. The configuration for this circuit includes all configs described above and takes inputs of the two execution traces ($\mathcal{L}$ and $\mathcal{L}$'), which is satisfied if the the memory described by the input execution trace is consistent.

[6]https://github.com/privacy-scaling-explorations/halo2/blob/main/halo2_proofs/examples/shuffle_api.rs

```
pub(crate) struct SortedMemoryConfig<F: Field + PrimeField> {
    /// The fields of an execution trace
    pub(crate) trace_record: TraceRecordWitnessTable<F>,
    /// The difference between the current and the previous address
    pub(crate) addr_cur_prev: IsZeroConfig<F>,
    /// The config for checking the current address||time_log is bigger
    /// than the previous one
    pub(crate) greater_than: GreaterThanConfig<F, 6>,
    /// The selectors
    pub(crate) selector: Column<Fixed>,
    pub(crate) selector_zero: Selector,
    /// The lookup table
    pub(crate) lookup_tables: LookUpTables,
    /// Just the phantom data
    pub(crate) _marker: PhantomData<F>,
}
```

Listing 2: Configuration for checking constraints from the sorted execution trace.

```
pub(crate) struct ConsistencyConfig<F: Field + PrimeField> {
    // the config of the original memory
    pub(crate) original_memory_config: OriginalMemoryConfig<F>,
    // the config of the sorted memory
    pub(crate) sorted_memory_config: SortedMemoryConfig<F>,
    // the config of permutation check
    pub(crate) permutation_config: ShuffleConfig,
    _marker: PhantomData<F>,
}
...
pub(crate) struct MemoryConsistencyCircuit<F: Field + PrimeField + From<B256>> {
    /// input_trace: Array of trace records before sorting (sorted by time_log)
    pub(crate) input: Vec<TraceRecord<B256, B256, 32, 32>>,
    /// shuffle_trace: Array after permutations (sorted by address and time_log)
    pub(crate) shuffle: Vec<TraceRecord<B256, B256, 32, 32>>,
    /// A marker since these fields do not use trait F
    pub(crate) marker: PhantomData<F>,
}
```

Listing 3: The configuration for the memory consistency circuit.

Chapter 8

Evaluation

In this chapter, we provide performance results on the runtime of the implementation described in Chapter 7 (including the KZG commitment scheme and the memory consistency circuit). These results include running unit tests of the overall system and the runtime of the prover and the verifier in the memory consistency circuit.

All measurements shown in this section are performed on the hardware with specifications as follows:

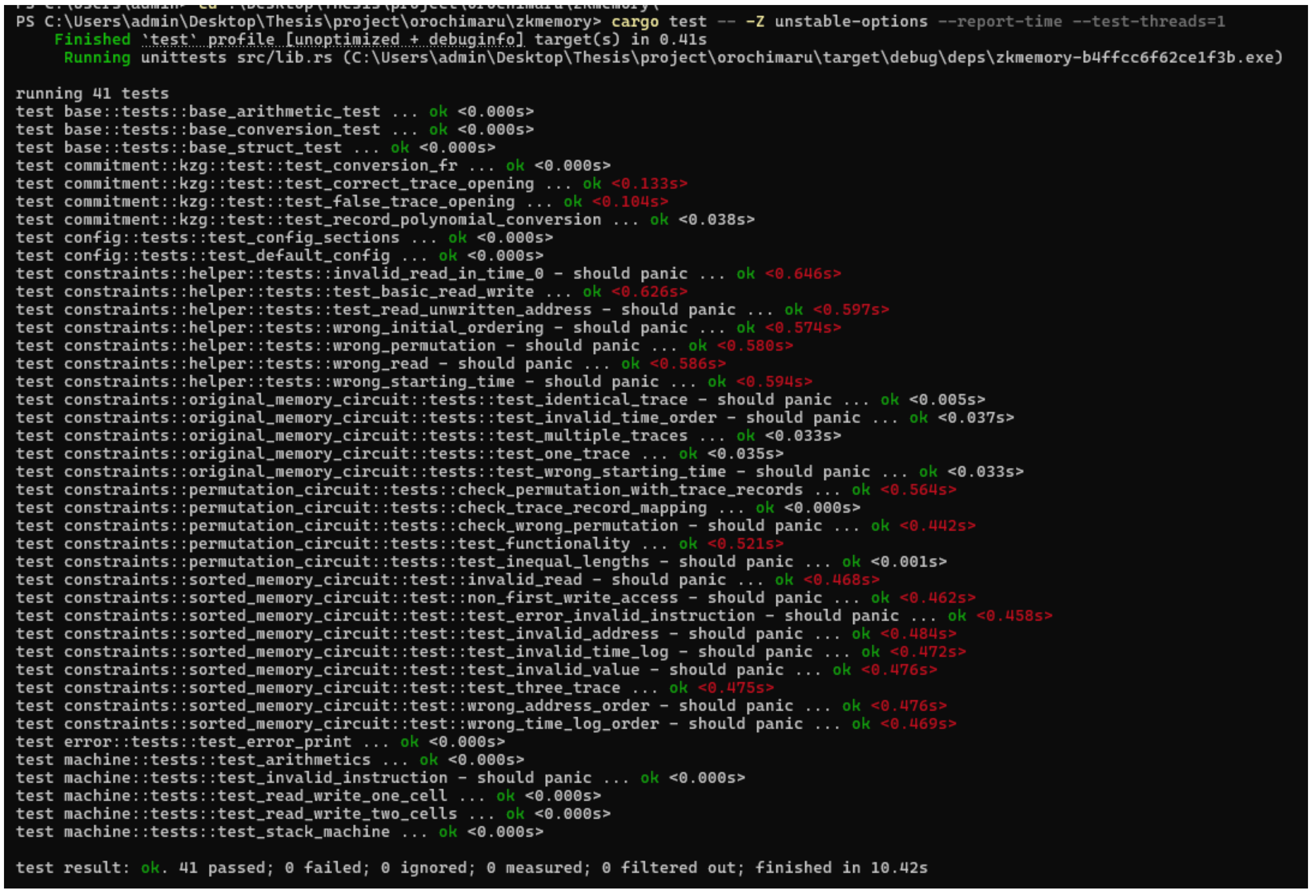
Processor	:	AMD Ryzen 7 6800H with Radeon Graphics
Architecture	:	x86/64
RAM size	:	16GB
Disk size	:	512 GB
Operating system	:	Microsoft Windows 11 Home Single Language.

8.1 Running Unit Tests

To ensure the proper functionality of core modules of the system, we list a set of unit tests the zkMemory implemented to cover the common functionalities of the module in Table 8.1. Note that module `helper` is a separate module that runs unit tests of module `memory_consistency_circuit`. The test result is shown in Figure 8.1.

ID	Module	Test name	Status
B01	base	base_arithmetic_test	Pass
B02	base	base_conversion_test	Pass
B03	base	base_struct_test	Pass
C01	config	test_default_config	Pass
C02	config	test_config_section	Pass
M01	machine	test_arithmetics	Pass
M02	machine	test_invalid_instruction	Pass
M03	machine	test_read_write_one_cell	Pass
M04	machine	test_read_write_two_cells	Pass
M05	machine	test_stack_machine	Pass
K01	kzg	test_conversion_fr	Pass
K02	kzg	test_correct_trace_opening	Pass
K03	kzg	test_false_trace_opening	Pass
K04	kzg	test_record_polynomial_conversion	Pass
O01	original_memory_circuit	test_one_trace	Pass
O02	original_memory_circuit	test_multiple_traces	Pass
O03	original_memory_circuit	test_invalid_time_order	Pass
O04	original_memory_circuit	test_identical_trace	Pass
O05	original_memory_circuit	test_wrong_starting_time	Pass
P01	permutation_circuit	check_permutation_with_trace_records	Pass
P02	permutation_circuit	check_trace_record_mapping	Pass
P03	permutation_circuit	check_wrong_permutation	Pass
P04	permutation_circuit	test_inequal_lengths	Pass
P05	permutation_circuit	test_functionality	Pass
S01	sorted_memory_circuit	invalid_read	Pass
S02	sorted_memory_circuit	non_first_write_access	Pass
S03	sorted_memory_circuit	test_invalid_instruction	Pass
S04	sorted_memory_circuit	test_invalid_address	Pass
S05	sorted_memory_circuit	test_invalid_time_log	Pass
S06	sorted_memory_circuit	test_invalid_value	Pass
S07	sorted_memory_circuit	wrong_time_log_order	Pass
S08	sorted_memory_circuit	wrong_address_order	Pass
S09	sorted_memory_circuit	test_three_traces	Pass
H01	helper	test_basic_read_write	Pass
H02	helper	invalid_read_in_time_0	Pass
H03	helper	test_read_unwritten_address	Pass
H04	helper	wrong_initial_ordering	Pass
H05	helper	wrong_permutation	Pass
H06	helper	wrong_read	Pass
H07	helper	wrong_starting_time	Pass

Table 8.1: List of unit tests for the modules.



```
PS C:\Users\admin\Desktop\Thesis\project\orochimaru\zkmemory> cargo test -- -Z unstable-options --report-time --test-threads=1
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.41s
     Running unittests src/lib.rs (C:\Users\admin\Desktop\Thesis\project\orochimaru\target\debug\deps\zkmemory-b4ffcc6f62ce1f3b.exe)

running 41 tests
test base::tests::base_arithmetic_test ... ok <0.000s>
test base::tests::base_conversion_test ... ok <0.000s>
test base::tests::base_struct_test ... ok <0.000s>
test commitment::kzg::test::test_conversion_fr ... ok <0.000s>
test commitment::kzg::test::test_correct_trace_opening ... ok <0.133s>
test commitment::kzg::test::test_false_trace_opening ... ok <0.104s>
test commitment::kzg::test::test_record_polynomial_conversion ... ok <0.038s>
test config::tests::test_config_sections ... ok <0.000s>
test config::tests::test_default_config ... ok <0.000s>
test constraints::helper::tests::invalid_read_in_time_0 - should panic ... ok <0.646s>
test constraints::helper::tests::test_basic_read_write ... ok <0.626s>
test constraints::helper::tests::test_read_unwritten_address - should panic ... ok <0.597s>
test constraints::helper::tests::wrong_initial_ordering - should panic ... ok <0.574s>
test constraints::helper::tests::wrong_permutation - should panic ... ok <0.580s>
test constraints::helper::tests::wrong_read - should panic ... ok <0.586s>
test constraints::helper::tests::wrong_starting_time - should panic ... ok <0.594s>
test constraints::original_memory_circuit::tests::test_identical_trace - should panic ... ok <0.005s>
test constraints::original_memory_circuit::tests::test_invalid_time_order - should panic ... ok <0.037s>
test constraints::original_memory_circuit::tests::test_multiple_traces ... ok <0.033s>
test constraints::original_memory_circuit::tests::test_one_trace ... ok <0.035s>
test constraints::original_memory_circuit::tests::test_wrong_starting_time - should panic ... ok <0.033s>
test constraints::permutation_circuit::tests::check_permutation_with_trace_records ... ok <0.564s>
test constraints::permutation_circuit::tests::check_trace_record_mapping ... ok <0.000s>
test constraints::permutation_circuit::tests::check_wrong_permutation - should panic ... ok <0.442s>
test constraints::permutation_circuit::tests::test_functionality ... ok <0.521s>
test constraints::permutation_circuit::tests::test_inequal_lengths - should panic ... ok <0.001s>
test constraints::sorted_memory_circuit::test::invalid_read - should panic ... ok <0.468s>
test constraints::sorted_memory_circuit::test::non_first_write_access - should panic ... ok <0.462s>
test constraints::sorted_memory_circuit::test::test_error_invalid_instruction - should panic ... ok <0.458s>
test constraints::sorted_memory_circuit::test::test_invalid_address - should panic ... ok <0.484s>
test constraints::sorted_memory_circuit::test::test_invalid_time_log - should panic ... ok <0.472s>
test constraints::sorted_memory_circuit::test::test_invalid_value - should panic ... ok <0.476s>
test constraints::sorted_memory_circuit::test::test_three_trace ... ok <0.475s>
test constraints::sorted_memory_circuit::test::wrong_address_order - should panic ... ok <0.476s>
test constraints::sorted_memory_circuit::test::wrong_time_log_order - should panic ... ok <0.469s>
test error::tests::test_error_print ... ok <0.000s>
test machine::tests::test_arithmetics ... ok <0.000s>
test machine::tests::test_invalid_instruction - should panic ... ok <0.000s>
test machine::tests::test_read_write_one_cell ... ok <0.000s>
test machine::tests::test_read_write_two_cells ... ok <0.000s>
test machine::tests::test_stack_machine ... ok <0.000s>

test result: ok. 41 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 10.42s
```

Figure 8.1: *Unit test result.*

8.2 KZG Commitment Scheme and Memory Consistency Circuit Performance

To evaluate the performance of the proposed KZG commitment scheme, we measure the performance of the four algorithms, namely Setup, CommitTrace, CreateTraceWitness (prover performance), and $VerifyTraceWitness$ (verifier performance). Since we are using the BN256 curve for our commitment scheme, the proof size is the size of a group element, which is 256 bits corresponding to the size of the underlying prime field of the curve. Table 8.2 depicts all other performance results. As we can see, the scheme takes up the most time in the initialization process to compute the public key (which consists of two common reference strings). After the initialization process, the commit performance, as well as prover and verifier times, are faster. All processes in the commitment scheme are done relatively fast since we fix the maximum degree of the polynomials to seven.

Table 8.2: Evaluation of the proposed commitment scheme.

Setup	CommitTrace	CreateTraceWitness	VerifyTraceWitness	Proof size
$33.9938ms$	$10.0455ms$	$23.5155ms$	$29.7284ms$	256

Figure 8.2 depicts the prover and verifier time of the memory consistency circuit with respect to the size of the RAM program. As in this plot, the prover time increases linearly with the size of the instructions. The verifier time also has a similar trend, except for the fluctuation in the middle since, at these points, the size of the instructions is not greatly different, while there may be a fluctuation in the number of WRITE and READ instructions in the program.

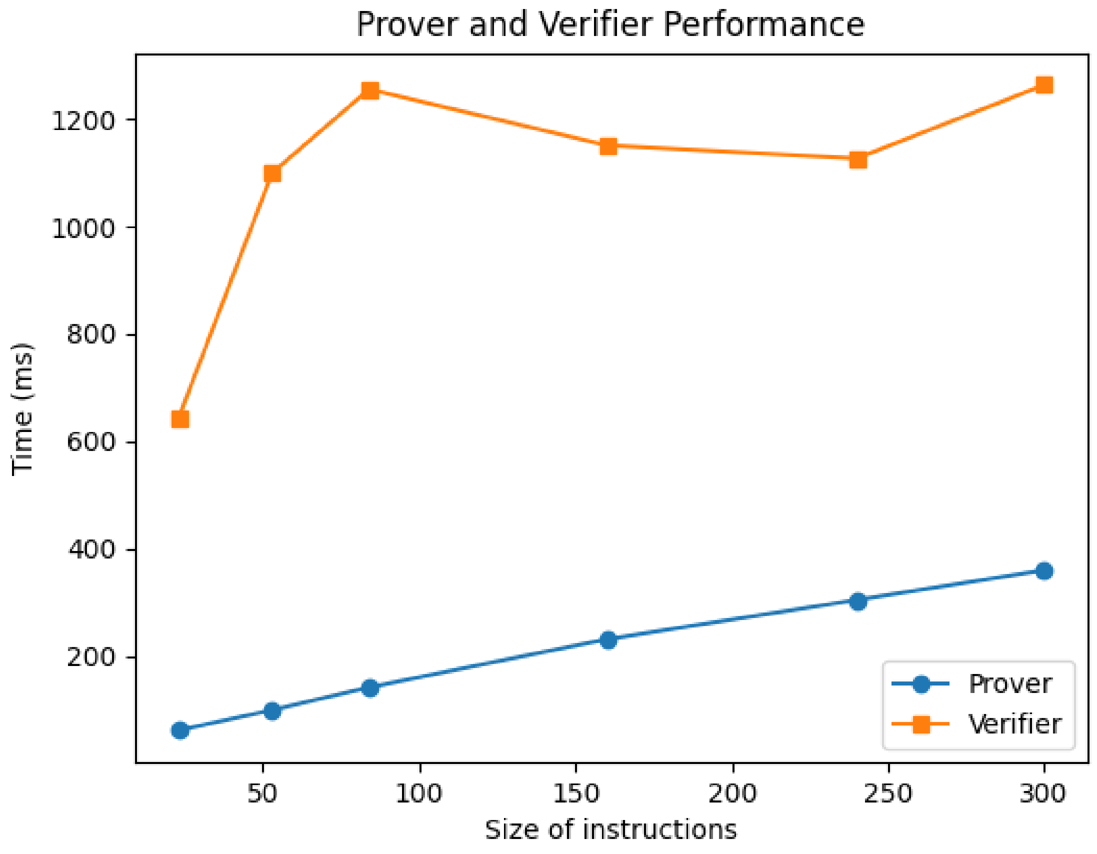


Figure 8.2: *Prover and Verifier performance of consistency check.*

Chapter 9

Conclusion

9.1 Results

In this project, we proposed to use a generalization of the memory of a RAM program from Orochi Network[1], including the memory layout in Section 5.1, basic memory access instructions in Section 5.2, and a definition of the execution trace in Section 5.3.1. We also proposed to use two existing commitment schemes for committing the execution trace, namely, Merkle Tree in Section 5.3.2, and KZG Polynomial Commitment Scheme in Section 5.3.3. In the KZG Polynomial Commitment Scheme, we employed multiproofs for multiple evaluations of the polynomial. This promotes the advantages of the KZG Polynomial Commitment Scheme: constant proof size and verifier's verification time (even for multiproofs). We also proposed a constraint system based on the work of Franzese et al. [FKL+21] that can prove the consistency of the memory in a RAM program in zero knowledge in Section 6.2. We also gained an enormous amount of background knowledge in algebra and elliptic curves, as well as many concepts of cryptography, zero-knowledge, RAM programs, and state-of-the-art works on RAM programs.

9.2 Future Works

In the future, we aim to achieve the following goals:

- We aim to define more instructions based on the 4 basic ones (arithmetic instructions, load/save instructions between registers and the memory storage, and instructions

[1] `https://github.com/orochi-network/orochimaru/blob/main/zkmemory/README.md`

between registers and registers) and propose a constraint system for all instructions.

- Research and implement Verkle Tree with KZG Polynomial Commitment Scheme: Verkle Tree is a k-ary tree that can provide shorter paths than Merkle Tree. We can achieve constant proof size and verification time when employing the KZG Commitment Scheme to Verkle Tree.
- Research a solution to prove the correct execution of a RAM program in zero-knowledge.

Bibliography

[AAEHR22] Sherif Abdelhaleem, Salwa Abd-El-Hafiz, and Ahmed Radwan. A generalized framework for elliptic curves based prng and its utilization in image encryption. *Scientific Reports*, 12, 08 2022.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2087–2104. ACM, 2017.

[Bar16] Elaine Barker. Recommendation for key management, part 1: General, 2016-01-28 2016.

[BCG$^+$13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013.

[BCG$^+$18] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. Nearly linear-time zero-knowledge proofs for correct program execution. Cryptology ePrint Archive, Paper 2018/380, 2018. https://eprint.iacr.org/2018/380.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin

Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 781–796. USENIX Association, 2014.

[BDFG20] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Efficient polynomial commitment schemes for multiple points and polynomials. Cryptology ePrint Archive, Paper 2020/081, 2020. https://eprint.iacr.org/2020/081.

[BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. *IACR Cryptol. ePrint Arch.*, page 1021, 2019.

[BHR$^+$20] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Public-coin zero-knowledge arguments with (almost) minimal time and space overheads. Cryptology ePrint Archive, Paper 2020/1425, 2020. https://eprint.iacr.org/2020/1425.

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018. https://eprint.iacr.org/2018/046.

[BSCG$^+$20] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Tinyram architecture specification v2.000, 2020.

[BSCR$^+$18] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for r1cs. Cryptology ePrint Archive, Paper 2018/828, 2018. https://eprint.iacr.org/2018/828.

[BSCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. Cryptology ePrint Archive, Paper 2014/595, 2014. https://eprint.iacr.org/2014/595.

[Car09] Barbara Carminati. *Merkle Trees*, pages 1714–1715. Springer US, Boston, MA, 2009.

[CGS$^+$23] Zhixi Cai, Shreya Ghosh, Kalin Stefanov, Abhinav Dhall, Jianfei Cai, Hamid Rezatofighi, Reza Haffari, and Munawar Hayat. Marlin: Masked autoencoder for facial video representation learning, 2023.

[Dam00] Ivan Damgård. Efficient concurrent zero-knowledge in the auxiliary string model. In *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 418–430. Springer, 2000.

[dSGOTV22] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of RAM programs from any public-coin zero-knowledge system. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks - 13th International Conference, SCN 2022, Amalfi, Italy, September 12-14, 2022, Proceedings*, volume 13409 of *Lecture Notes in Computer Science*, pages 615–638. Springer, 2022.

[EA23] Alex Evans and Guillermo Angeris. Succinct proofs and linear algebra. Cryptology ePrint Archive, Paper 2023/1478, 2023. https://eprint.iacr.org/2023/1478.

[FKL+21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for RAM programs. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 178–191. ACM, 2021.

[FNO14] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. Cryptology ePrint Archive, Paper 2014/598, 2014. https://eprint.iacr.org/2014/598.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 291–304. ACM, 1985.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021. https://eprint.iacr.org/2021/1063.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.

[GW20a] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Paper 2020/315, 2020. https://eprint.iacr.org/2020/315.

[GW20b] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Paper 2020/315, 2020. https://eprint.iacr.org/2020/315.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, page 953, 2019.

[HK20] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. Cryptology ePrint Archive, Paper 2020/136, 2020. https://eprint.iacr.org/2020/136.

[HK22] David Heath and Vladimir Kolesnikov. A 2.1 khz zero-knowledge processor with bubbleram. Cryptology ePrint Archive, Paper 2022/809, 2022. https://eprint.iacr.org/2022/809.

[HMR15] Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 150–169. Springer, 2015.

[HPS08] J. Hoffstein, J. Pipher, and J.H. Silverman. *An Introduction to Mathematical Cryptography*. Undergraduate Texts in Mathematics. Springer, 2008.

[HYDK22] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast zk processor with cached ram for ansi c programs. Cryptology ePrint Archive, Paper 2022/810, 2022. https://eprint.iacr.org/2022/810.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: How to prove non-algebraic statements efficiently. Cryptology ePrint Archive, Paper 2013/073, 2013. https://eprint.iacr.org/2013/073.

[JKPT12] Abhishek Jain, Stephan Krenn, Krzysztof Pietrzak, and Aris Tentes. Commitments and efficient zero-knowledge proofs from learning parity with noise. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 663–680. Springer, 2012.

[KMS$^+$15] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. Cryptology ePrint Archive, Paper 2015/675, 2015. https://eprint.iacr.org/2015/675.

[KZG10] Kate, Gregory M. Zaverucha, and Ian Goldberg. Polynomial commitments * aniket, 2010.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings. Cryptology ePrint Archive, Paper 2019/099, 2019. https://eprint.iacr.org/2019/099.

[MRS17] Payman Mohassel, Mike Rosulek, and Alessandra Scafuro. Sublinear zero-knowledge arguments for RAM programs. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 501–531, 2017.

[Nit20] Anca Nitulescu. zk-snarks: A gentle introduction. 2020.

[Ped92] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[PFM$^+$22] Luke Pearson, Joshua Fitzgerald, Héctor Masip, Marta Bellés-Muñoz, and Jose Luis Muñoz-Tapia. Plonkup: Reconciling plonk with plookup. Cryptology ePrint Archive, Paper 2022/086, 2022. https://eprint.iacr.org/2022/086.

[PGHR13] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. Cryptology ePrint Archive, Paper 2013/279, 2013. https://eprint.iacr.org/2013/279.

[Set20] Srinath T. V. Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 704–737. Springer, 2020.

[Sho05] Victor Shoup. *A computational introduction to number theory and algebra.* Cambridge University Press, 2005.

[TK22] Duc A. Tran and Bhaskar Krishnamachari. Blockchain in a nutshell. *CoRR*, abs/2205.01091, 2022.

[WSR$^+$15] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015.* The Internet Society, 2015.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1074–1091. IEEE, 2021.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2986–3001. ACM, 2021.