PRIVATE AND VERIFIABLE COMPUTATION

by

Dimitris Mouris

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Winter 2024

PRIVATE AND VERIFIABLE COMPUTATION

by

Dimitris Mouris

Approved: _____
Jamie Phillips, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____
Levi T. Thompson, Ph.D.
Dean of the College of Engineering

Approved: _____
Louis F. Rossi, Ph.D.
Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Nektarios G. Tsoutsos, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Rudolf Eigenmann, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Xiaoming Li, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Rui Zhang, Ph.D.
Member of dissertation committee

VITA

Dimitris Mouris received his B.Sc. (2016) and M.Sc. (2018) degrees in computer science and computer systems, respectively, from the National and Kapodistrian University of Athens, Greece. In 2019, Dimitris started his Ph.D. in electrical and computer engineering at the University of Delaware, Newark, DE, USA, and passed the qualification exam in the Summer of 2019. He also submitted his dissertation proposal in the Spring of 2023. Since 2019, Dimitris has been a research and teaching assistant at the University of Delaware. His research is in the intersection of applied cryptography and privacy, with a special interest in the area of privacy-enhancing technologies using secure multi-party computation, zero-knowledge proofs, and homomorphic encryption. He has authored multiple articles in transactions and conference proceedings.

Apart from his academic experience, Dimitris has spent time working in the industry. In the summer of 2022, Dimitris interned at Meta Inc. as a Research Scientist in the Statistics & Privacy team and worked on private advertising and private record linkage protocols. After his summer internship, he continued working for three months as a part-time student researcher to publish his internship work. In the summer of 2021, Dimitris interned at Amazon Web Services (AWS) and worked on enabling role-based access control for Amazon Redshift, a data analytics warehouse. His work has been productionized allowing millions of customers to have better control over permissions and security privileges. Before joining the University of Delaware in 2019, Dimitris worked as a research assistant at the Athena Research & Innovation Center in the European Union's Horizon 2020 "My Health My Data" project. He designed and implemented an end-to-end framework for privacy-preserving medical data analytics using secure multi-party computation.

Dimitris participates in the Crypto Forum Research Group (CFRG) of the Internet Engineering Task Force (IETF) to enable real-world use cases of secure computation. Parts of his work have been used by CFRG specifications. He has also contributed to the Open Quantum Safe (OQS) project, an open-source project that aims to support the development and prototyping of quantum-resistant cryptography. From 2020 until 2022, Dimitris was also the global challenge co-lead of the international Embedded Security Challenge (ESC) competition; an annual student-run competition during the Cyber Security Awareness Worldwide (CSAW) event.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deep gratitude to my advisor, Nektarios Tsoutsos, for his unconditional support and continuous guidance throughout my academic journey. Nektarios' mentoring and always optimistic outlook have helped me grow not only as a researcher but also as an individual. I first met Nektarios in 2016 while he was pursuing his Ph.D. under Mihalis Maniatakos' guidance. During that time, I was completing my Master's at the University of Athens and they gave me the opportunity to collaborate on a research project in secure computation. This was my first exposure to the field of Cryptography and research in general, and they were an invaluable resource of guidance. I would like to express my gratitude to both Nektarios and Mihalis for their mentorship at that time. A year later, Nektarios joined the faculty at the University of Delaware and we were both excited to continue working together.

I would also like to thank my committee members, Rudolf Eigenmann, Xiaoming Li, and Rui Zhang for their valuable insights towards completing this dissertation. I am also grateful to my awesome internship mentors Shubho Sengupta, Prasad Buddhavarapu, Ni Trieu, Daniel Masny, and Benjamin Case from the Statistics and Privacy team at Meta, as well as Pavel Sokolov and Huiyuan Wang from the Amazon Redshift Catalog (RedCat) team. I had the privilege of spending two incredible summers working with them, and they helped me develop both as a researcher and as an engineer. I want to give special acknowledgments to my friends and co-authors, Chaz Gouert and Pratik Sarkar, for the multiple brainstorming sessions and the great moments we shared that led to the realization of this work. Furthermore, I am grateful to my all friends for being there for me throughout this journey.

My sincere thanks to my best friend and wife, Eirini Vangeli, for her unlimited love and for always being on my side. Moving from Greece to the United States was a big decision but she embraced it and pursued her own dreams. Together, we have explored a new country, created a new home, and continue chasing our dreams. My deepest thanks to my parents for their unconditional love and for opening so many doors for me to achieve my dreams. Without their endless support, this journey would not have been possible.

At last, my work was made possible by the support of the University of Delaware Research Foundation (Grant #21A01012), the National Science Foundation (Grants #2239334 and #1931916), Discover Bank and the Arkansas Venture Center, the US Department of Energy (Grant #DE-EE0008768), and the Electrical and Computer Engineering department at the University of Delaware.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The rise of cloud computing and big data analytics offers significant benefits for individuals and organizations as they enable a plethora of applications in diverse fields such as healthcare, home automation, and many more. These technologies enable individuals and organizations to adjust computational resources effortlessly and take advantage of large amounts of data, leading to improved efficiency and reduced costs. Unfortunately, processing vast amounts of data increases the risk of data theft and misuse as sensitive information is accessible by the cloud provider and vulnerable to attacks from third parties. Zero-knowledge proofs (ZKP), secure multiparty computation (MPC), and homomorphic encryption (HE) are key cryptographic techniques that focus on protecting data confidentiality while enabling valuable computations to be performed on sensitive data. Unfortunately, generic solutions incur significant performance overheads and may not be practical for many real-world use cases; thus, specialized protocols need to be devised. Another challenge with HE and MPC is to provide verifiable guarantees on the integrity of the computation, i.e., that it was performed faithfully. ZKPs offer a solution to this problem, yet combining these technologies requires intricate solutions.

This dissertation focuses on private and verifiable computation. In particular, we start by introducing the Zilch framework for developing transparent ZKPs, i.e., ZKPs that do not need a trusted setup. Zilch consists of a back-end that allows verifying MIPS-like instructions and a front-end that compiles high-level code to our zero-knowledge MIPS back-end. As a result, our framework makes ZKPs more accessible and can facilitate many real-world applications. More specifically, we continue this dissertation by utilizing Zilch to create specialized protocols for proving both functional and security properties of intellectual property (IP) netlists without revealing

anything about them. These works focus on thwarting IP piracy in the integrated circuit industry as IP vendors want to convince IP buyers about various properties of their netlists while still maintaining the privacy of their designs.

Finally, we focus on privacy-preserving and verifiable statistics based on inputs from multiple clients. More specifically, we introduce two protocols that offer verifiable guarantees of the correctness of the final result and are secure against participants who do not follow the protocol specification. Both these works assume a set of clients that hold some private inputs and some aggregation servers that wish to compute statistics based on the client inputs privately. The first work is called Masquerade and focuses on aggregations and histograms, while the latter, called PLASMA, focuses on more elaborate statistics such as private heavy-hitters (i.e., finding the most popular client inputs). PLASMA relies on three servers and offers even stronger security guarantees by considering that even one of the servers may be malicious and not follow the protocol specification. These two works showcase real-world applications of private and verifiable computation.

Chapter 1

INTRODUCTION

The increasing digitization of information through the use of edge devices has opened up new opportunities for businesses to provide real-time and personalized services to individuals. This is achieved by collecting and transmitting data to cloud servers for analysis and computation. For example, using real-time location data, applications can suggest popular restaurants nearby, provide navigation routes with less congestion by combining data from multiple users, as well as help businesses identify crowded shopping areas and target their marketing and advertising efforts. Businesses also take advantage of cloud computing and big data for performing analytics, data backups, and scaling resources on demand. This allows the party that owns the data to learn useful statistics and insights about their data, while the cloud performs the heavy computational work and returns the results.

Although cloud computing offers great flexibility for multiple industries, delegating the computation of sensitive user data to untrusted cloud servers comes with many risks [223]. For one, cloud servers might run unreliable software that is susceptible to a wide range of cyberattacks or even run on unverified – or malicious – hardware [227, 228]. Even worse, cloud service providers might have monetary incentives to look into, store, and sell user data.

Furthermore, it is crucial to have verifiable guarantees in order to ensure the integrity of data and the correctness of the outsourced computation. Organizations and individuals need to be able to verify that the cloud computation has not been tampered with (i.e., that the results are trustworthy). Without verifiable integrity guarantees, there is a risk that the data and computations may be compromised, leading to inaccurate or biased results. The need for verifiability becomes even more critical

when dealing with sensitive or confidential data, such as personal or financial information. Additionally, verifiability is also important for compliance with regulations and standards, such as GDPR [234] and HIPAA [1].

In order to mitigate all these risks, researchers have developed a set of cryptographic techniques, called Privacy Enhancing Technologies (PETs), that enable applications in statistics, machine learning, and finance, to name a few, while protecting the privacy of individuals and organizations. These technologies enable organizations to perform useful computations on sensitive data without revealing the actual data to any of the parties involved. Before we delve into the problem statement of this dissertation, we first need to discuss the different PETs, the applications they enable, and their limitations.

1.1 Privacy-Enhancing Technologies (PETs)

Multiple applied cryptography techniques have been proposed to protect the privacy of user data and promote transparency in outsourced computation without relying on trusted third parties. Below we focus on the three more prominent techniques, which provide the pillars of my research.

Homomorphic Encryption (HE) is a natural fit for cloud computing by allowing operations to be performed on encrypted data. Users can encrypt their data on their devices, send it to the cloud for computation, and receive an encrypted output which only they can decrypt to access the final result. There are three types of homomorphic encryption: partial HE (PHE), leveled HE (LHE), and fully HE (FHE). PHE [186,194] enables only certain operations over encrypted data (i.e., either encrypted addition or multiplication but not both), while LHE enables any type of encrypted operation but only a limited number of times [57, 73, 115]. Finally, FHE [78, 114, 201] supports unlimited additions and multiplications but incurs significant overheads compared to PHE and LHE. There is a big line of research for accelerating FHE and making it more practical [77, 78, 131, 143] but its practicality is questionable in many real-world scenarios. A notable application of HE includes privacy-preserving machine learning

(ML) inference, where the cloud has a trained model with private weights and the users can upload their sensitive data for classification while keeping it private from the cloud [44, 79, 91, 97, 108, 152]. For example, a user can get a quick diagnosis by uploading an encrypted X-ray image of their lungs to a cloud service provider that runs an ML algorithm, and the cloud classifies it as either "healthy" or not – without being able to look at the image.

Secure Multi-Party Computation (MPC) enables multiple entities to jointly perform a computation without disclosing any individual's private inputs [22,120,172,246]. There are many MPC protocols that consider various threat assumptions about how the participants are behaving that result in security/performance trade-offs. Most practical protocols assume semi-honest (or passive) adversaries, meaning that they will follow the protocol specification as opposed to malicious (or active) adversaries. Another consideration is the number of honest participants and how privacy and correctness can be maintained when parties collude. MPC protocols that assume that the majority of the parties behave honestly (i.e., honest majority) typically utilize secret sharing as a basic tool [26,211]. A (t, n)-secret sharing scheme allows splitting a secret amongst n parties, so that t or more parties can reconstruct the secret while having less than t shares does not reveal anything about the secret. In this case, the parties first represent the function as a Boolean or arithmetic circuit and then each party shares its input with the other parties using secret sharing. The parties can now jointly evaluate the circuit gate by gate and finally reconstruct the shares on the output wires, which represent the output of the function that was encoded as the circuit. On the other hand, assuming a dishonest majority MPC is significantly more challenging and requires specialized solutions such as garbled circuits [23,247], GMW oblivious transfer [120,136], cut-and-choose [165], SPDZ [89], MPC in the head [137], and others [104].

Contrary to HE, most MPC protocols assume that the data owners participate in the execution of the protocol. However, this is not always the case; separating the MPC computing parties and the data owners brings MPC closer to the cloud paradigm. Notable examples include privacy-preserving data sharing and analytics,

as well as privacy-preserving machine learning (ML) based on sensitive user data. In the former case of analytics, MPC can be used to compute advanced statistics over the joint data of multiple participants without revealing the actual data to any of the parties, while in the case of privacy-preserving machine learning, MPC can be used to train ML models on private data without revealing the actual data to any of the parties. Additionally, similar to HE, MPC can be used for ML inference where the MPC computing parties act as "the cloud" and the users securely share their data and get back an encrypted classification.

Zero-Knowledge Proofs (ZKP) allow a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that a public statement is true without revealing anything else apart from the fact that this specific statement is true. Furthermore, ZKPs enable $\mathcal{P}$ to prove knowledge of some secret $\mathtt{w}$ without revealing it [54, 121, 122]. ZKPs have multiple practical applications including finance, where $\mathcal{P}$ can prove that a transaction is valid without revealing the details of the transaction (e.g., prove that they have enough funds to make a purchase without revealing their entire financial history to the seller, prove eligibility for a mortgage by showing that their salary is above a threshold without revealing it, etc.), and electronic voting (e-voting) where voters can prove that their vote was cast without revealing how they voted, and others.

ZKP constructions can be classified into three different categories based on their setup process: those with trusted setup per computation, those with transparent setup, and those with universal trusted setup. ZKPs that require a trusted setup per computation achieve a succinct/constant-size proof, which renders them ideal for blockchain applications, like Zerocash [32]. Such constructions are based on quadratic arithmetic programs [37, 113, 127, 195]. ZKPs with a universal trusted setup perform only one setup but can be used to prove multiple statements [65, 128, 173, 245]. Such techniques incur bigger proofs and slower verification compared to the first category. Unfortunately, both aforementioned categories rely on a secret randomness for their trusted setup (referred to as "toxic waste") that has to be deleted to prevent malicious parties from forging false proofs. Lastly, ZKPs with a transparent setup do not require

a trusted setup phase [31, 34, 35, 51, 61] as any randomness used is public coins.

Finally, although the focus of this dissertation is on ZKPs, HE, and MPC, we mention two more PETs worth noting for completeness: *Trusted Execution Environments (TEE)* and *Differential Privacy (DP)*. TEEs provide secure execution through specialized hardware and software; they create an isolated environment, known as an enclave, for a process to run and be invisible to other processes and even the operating system [14, 252]. On the other hand, DP focuses on an information-theoretic notion of output privacy and guarantees that the presence or absence of a single record in a database should not affect the query result, and thus provides privacy [100].

1.2 Problem Statement

It has become evident that ZKP, HE, and MPC comprise a powerful arsenal of techniques for private and trustworthy computation. Unfortunately, these techniques are often a double-edged sword: for one, they are complicated to use, especially for non-crypto-savvy developers. Utilizing some of these techniques requires expressing a computation as a polynomial, using commitment schemes, finite field arithmetic, and many other mathematical and cryptographic tools that many developers are not familiar with. Therefore, it is crucial to develop tools and general-purpose frameworks that render PETs easier to program and to adapt into existing applications. For another, they introduce computational overheads (compared to plaintext computation) and in some cases, they may even become impractical. Thus, specialized solutions need to be developed for certain applications in order to be feasible.

One such application arises from the Integrated Circuits (IC) industry. As ICs play a crucial role in most electronic devices, the security of IC designs is a top priority in today's highly interconnected global economy [203]. The IC supply chain involves acquiring Intellectual Property (IP) cores from third-party vendors (3PIP) and combining them with in-house designs to manufacture the System-on-Chip (SoC) in order to fabricate the IC [203, 204]. The increased IC demand has led to an increase in the number of 3PIP vendors that aim to increase their profits by offering reusable IP

cores, such as digital signal processors (DSPs) and FFT engines, that can be utilized by multiple design layouts [225]. Unfortunately, the widespread use of third-party IPs has attracted malicious actors that attempt to steal the IPs for financial gain [205] or use them to exfiltrate sensitive information. These malicious entities rely on attacks such as system-level analysis [226], reverse engineering [69], as well as hardware Trojans that are triggered under certain conditions (e.g., user input, time-based, etc.) [224] or are always on [146]. When activated, Trojans can alter device functionality by influencing output wires or creating a side channel through which sensitive data can be leaked. As a result, addressing IP piracy and prioritizing security in the IC supply chain is becoming a crucial concern.

IP core verification is a crucial step of SoC design [95], where the IP consumers provide functional requirements to the 3PIP vendors and the latter design circuits that meet these specifications. The correctness of 3PIP designs should be verified prior to integration, presenting a challenge for 3PIP vendors since they have to prove the functionality of their designs to system integrators while protecting the privacy of the circuit implementations. Ensuring that the circuit meets the specified requirements while being highly testable is crucial in the IC supply chain. Unfortunately, previous solutions focus only on functional verification and completely dismiss the privacy and the security of IP designs, leading to IP piracy. In particular, 3PIP vendors cannot demonstrate the functionality of the generated IP design to the system integrators (i.e., IP consumers) without revealing their IP, while on the other hand, IP consumers cannot test the netlist without having access to it. Thus, it is critical to eliminate this deadlock and enable IP verification both in terms of functionality (e.g., area, power, frequency) and in terms of security (e.g., the IP is Trojan-free) without disclosing the design of the IP netlist.

Zero-knowledge proofs offer a promising solution to this problem, where the 3PIP vendors can prove various properties of their private netlists without revealing any information about them. The ZKPs need to be transparent (i.e., no requirement for a trusted third party to perform the setup) so that the system integrators have provable

guarantees that the 3PIP vendors have not forged false proofs. In this dissertation, we propose new theoretical and technical solutions to solve this deadlock and thwart IP piracy.

Another such application focuses on privacy-preserving analytics and crowdsourcing based on sensitive data from multiple individuals and organizations while protecting the privacy of each participant. This is crucial in today's technology-driven world, where companies collect large amounts of user data for analysis (e.g., histograms, heavy-hitters, etc.) and improvement purposes. A notable example is smart metering, which can be applied to multiple participating households. Smart meters communicate the electrical usage almost in real-time to provide better system monitoring and customer billing than traditional meters. However, there are inherent privacy concerns since fine-grained measurements (e.g., one every 15 minutes) may reveal personal information about the number of people in a household and their activities [70, 90]. Different examples of privacy-preserving statistics and analytics include providing restaurant recommendations, traffic avoidance (i.e., different roads may have different densities – highways are busier than smaller suburban roads), as well as advertising (e.g., businesses can identify crowded shopping areas and target their marketing efforts) while preserving the privacy of the participants.

Existing PETs enable computing some of the aforementioned examples by allowing participants to encrypt their data and submit it to one or more servers that run MPC or HE. Unfortunately, generic solutions suffer from high overheads and most importantly, they do not verify the participants' encrypted inputs, which allows malicious parties to tamper with the encrypted computation. Therefore, it is of utmost importance to devise new optimized protocols tailored for real-world privacy-preserving analytics while verifying participants' inputs.

1.3 Contributions

This dissertation focuses on verifiable and private computation. An overview is presented in Fig. 1.1. In particular, Chapter 3 introduces a framework that accelerates

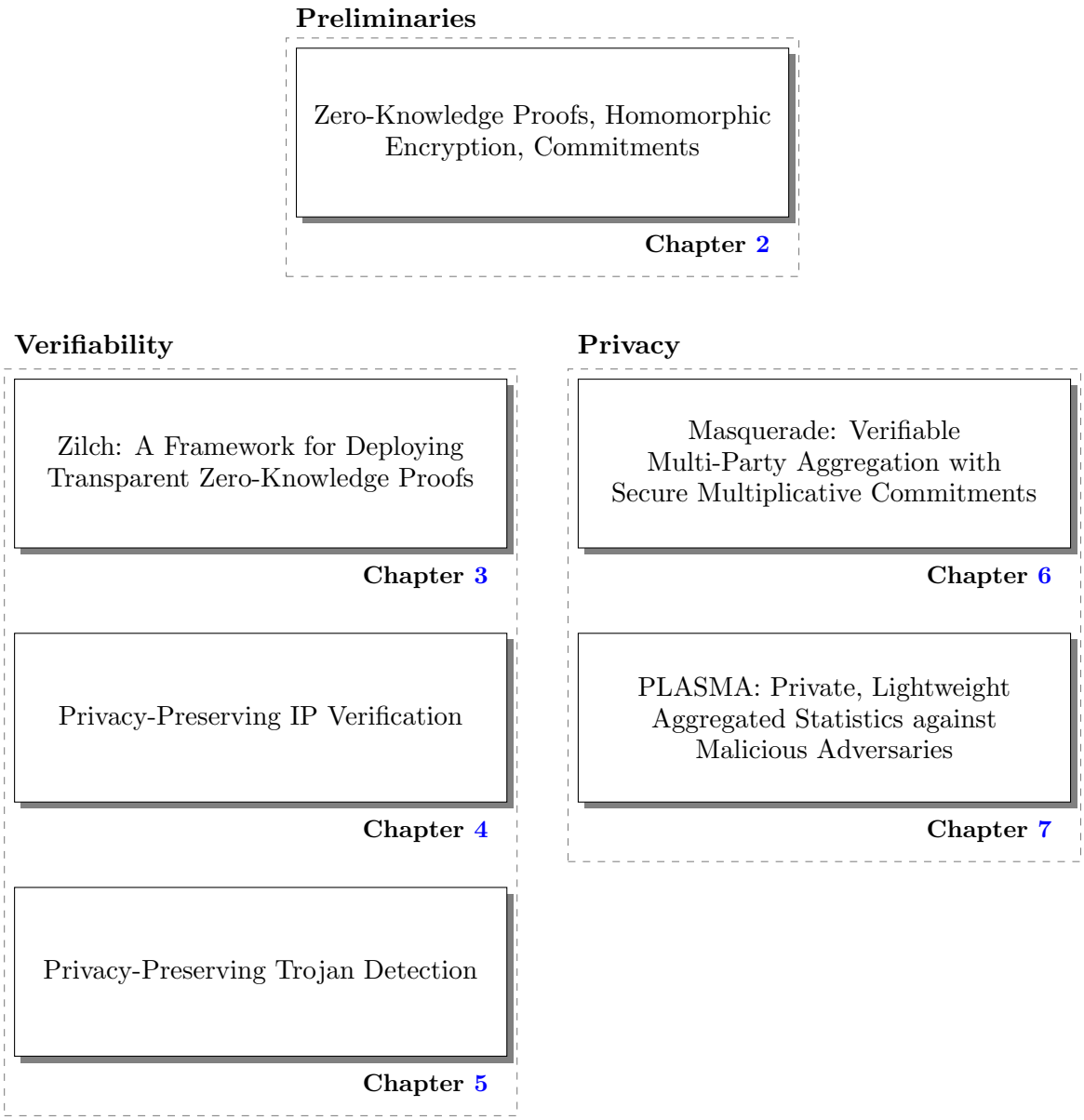


Figure 1.1: Overview of this dissertation.

and simplifies the deployment of ZKPs for any application transparently, i.e., without the need for a trusted setup. To foster usability, our framework incorporates a novel cross-compiler from an object-oriented Java-like language tailored to ZKPs as well as a powerful API that enables integration of ZKPs within existing C/C++ programs. Chapters 4 and 5 introduce Pythia and zk-Sherlock, two specialized ZKP frameworks that solve a major problem in the globalized integrated circuit (IC) supply chain. Both

these works utilize the Zilch framework as a back-end for ZKPs. Pythia allows third-party intellectual property (3PIP) vendors to prove to system integrators the functional properties of their circuit designs while protecting the privacy of the circuit implementations. On the other hand, zk-Sherlock enables 3PIP vendors to prove an intellectual property (IP) design is free of hardware Trojans (i.e., attacks that alter device functionalities when triggered or create a side channel to leak sensitive information such as cryptographic keys) without disclosing the corresponding netlist. To remain hidden, some of these Trojans are "rarely activated", meaning that for most common inputs there is no switching activity in the area of the circuit where the Trojan lies. This research direction takes advantage of this observation to simulate the circuit with multiple random input patterns and keep track of the switching activity throughout the whole circuit. Detecting switching activity in all the gates of the circuit with a small number of inputs indicates the absence of Trojans, whereas if there is a part of the circuit that does not have any switching activity even after a relatively large number of random inputs, we mark that as a potential culprit. Working together, these two frameworks can mitigate the threat of IP reuse and piracy in the IC industry.

Next, in Chapters 6 and 7, we introduce two protocols for computing privacy-preserving statistics. The former (i.e., Chapter 6), describes Masquerade, a specialized protocol for computing private statistics such as aggregations, means, and histograms without revealing anything about participants' data. Masquerade privately verifies the validity of shared data points to ensure the integrity of the statistics against untrusted participants and provides public verifiability. The latter (i.e., Chapter 7), presents a novel protocol, called PLASMA, for finding the most popular participant data in a privacy-preserving way. This problem is known as computing the "heavy-hitters". PLASMA achieves full security against the collusion of a malicious server and malicious clients, i.e., the privacy of participants is maintained even if one of the servers colludes with multiple clients and deviates from the protocol specification. Additionally, even in that scenario, PLASMA guarantees the correctness of the output by allowing the servers to non-interactively verify client inputs and preemptively reject malformed ones.

1.3.1 General-Purpose Transparent Zero-Knowledge Proofs

One of our first works introduces Zilch, a framework for developing transparent zero-knowledge proofs [184]. While state-of-the-art ZKP protocols rely on arithmetic circuits that need to be regenerated for each different computation, in Zilch we have implemented a MIPS-like processor model that allows verifying each instruction independently and composing a proof for the execution of the target application. Our ZKP-tailored processor, called zMIPS, uses instruction sequences rather than static arithmetic circuits. To further foster usability, Zilch incorporates a novel compiler from our customized object-oriented Java-like language, called ZeroJava, to zMIPS instructions. Finally, Zilch exposes a powerful API that enables the integration of ZKP within existing C/C++ programs. We demonstrate the flexibility and ease of use of Zilch using two real-life applications (i.e., the first focuses on secure auctions, while the second on range proofs).

1.3.2 Functional and Security Verification of Intellectual Property (IP)

Next, we focus on resolving the deadlock between third-party intellectual property vendors (3PIP) and IP consumers by introducing novel zero-knowledge constructions to prove various functional and security properties of netlists without disclosing anything about them. More specifically, in [178, 182] we introduce Pythia, a framework that enables 3PIP vendors to convince system integrators about various functional properties of a circuit (e.g., area, power, frequency) without disclosing its netlist (i.e., in zero-knowledge). Our approach comprises a circuit compiler that transforms arbitrary netlists into a zero knowledge-friendly format and a library of modules that provide cryptographic guarantees for various properties of the netlist while hiding the actual gates.

Furthermore, in zk-Sherlock [179] we extend this line of work to enable 3PIP vendors to convince system integrators that their IPs are free of hardware Trojans (i.e., they are safe to use), while also maintaining the privacy of their netlist designs. We use

a specialized circuit compiler that transforms arbitrary netlists into a zero-knowledge-friendly format and introduces a versatile Trojan detection module that maintains the privacy of the actual netlist. All these works (i.e., [178, 179, 182]) utilize the Zilch framework [184] as a back-end for ZKPs.

1.3.3 Privacy-Preserving Analytics

Next, we focus on protocols for privacy-preserving statistics and analytics on data of multiple individuals and organizations while providing verifiable guarantees of the correctness of the computation. More specifically, our work in Masquerade [183] computes private statistics, such as sum, average, and histograms without revealing anything about participants' data. An important aspect of any private computation over data of multiple entities is to provide provable guarantees on the integrity of data aggregations. To ensure the integrity of data aggregations, we propose a tailored multiplicative commitment scheme and publish all the participants' commitments on a ledger to provide public verifiability. Lastly, in any private crowd-sourcing protocol that assumes multiple participants, it is safe to assume that some of them may act maliciously and attempt to poison the aggregation results by submitting invalid inputs. We complement Masquerade with two ZKP protocols that detect malicious participants who attempt to poison the aggregation results by submitting invalid inputs. Masquerade ensures the validity of shared data points before being aggregated, enabling a broad range of applications.

Finally, we propose PLASMA [181], a protocol for privacy-preserving analytics such as heavy hitters and histograms that utilizes three data-collection servers. PLASMA allows computing private analytics while achieving full security (i.e., both privacy and correctness) against a collusion of a malicious server and malicious clients. Our framework allows each client to non-interactively send a message to the servers and then go offline. We introduce a novel primitive, called verifiable incremental distributed point function (VIDPF) that employs lightweight techniques based on efficient

hashing and allows the servers to non-interactively validate client inputs and preemptively reject malformed ones without learning anything about the clients' submissions (i.e., in zero-knowledge).

Chapter 2

PRELIMINARIES

2.1 Models of Computation

There exist many different models of computation, some less powerful yet simpler, while others are more sophisticated. In the context of this article, we delve into two models that enable the execution of arbitrary computer programs: Turing machines (TMs) and arithmetic circuits (ACs).

A Turing Machine is a model of computation that consists of an infinite tape, a tape head, and a finite table of rules. At each step, the tape head reads a symbol from the tape and determines which action to perform from the finite table and then either moves one cell to the left or right or halts the computation. This abstract machine, despite its simplicity, is capable of executing any algorithm given as a set of rules for an input provided in the tape [218]. A universal Turing machine (UTM) is a TM whose algorithm (table of rules) implements *a simulator* for any arbitrary TM with arbitrary input tape. A fundamental difference between a TM and a UTM is that the former is programmed with a rules table to evaluate a specific problem, while the latter works with the description of any TM and thus can evaluate any program.

An Arithmetic Circuit over a field $\mathbb{F}$ consists of input and output gates that are connected with intermediate gates through wires. The input values proceed through a sequence of gates performing either addition ($+$) or multiplication ($\times$); a simple example is illustrated in Fig. 2.1. Transforming certain classes of programs into ACs can be straightforward if they only involve the addition and multiplication of elements of the finite field. Notably, this approach is equivalent to the evaluation of polynomials

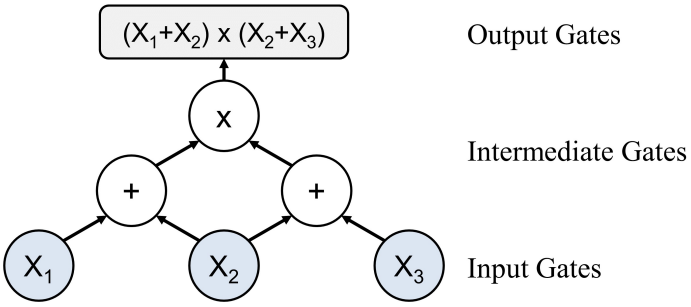


Figure 2.1: Example of a computation using arithmetic circuits. The output can be expressed using a polynomial expression.

over a field $\mathbb{F}$, so the outputs of the arithmetic circuit can be expressed as a set of polynomials over the input variables.

Turing machines and ACs are equivalent models of computation, i.e., given a program and an input, both models can compute the same output. In fact, any Turing machine can be unrolled into a circuit somewhat larger than the number of steps in the computation [241]. Our abstract machine in Zilch offers a more flexible model of computation since it is the equivalent of a UTM: its input is a program defined as a sequence of instructions that can consume any given input.

2.2 Principles of ZKPs and VC

2.2.1 Verifiable Computation

The typical scenario in VC is that the verifier ($\mathcal{V}$) sends a program description Ψ and an input $\mathbb{x}$ for that program to the prover ($\mathcal{P}$). Then, $\mathcal{P}$ computes and returns the output $\mathbb{y} = \Psi(\mathbb{x})$ of the execution of that program on input $\mathbb{x}$ to $\mathcal{V}$ along with a short proof which can be efficiently verified by $\mathcal{V}$. In this case, both parties express the computation Ψ as a set of constraints involving $\mathbb{x}$ and $\mathbb{y}$. Those constraints are essentially equations over a finite field $\mathbb{F}$ modulo a large prime. Consecutively, $\mathcal{P}$ solves the constraints (i.e., finds a satisfying assignment), where a solution exists if and only if $\mathbb{y} = \Psi(\mathbb{x})$. These constraints are equivalent to ACs [37, 195], where the gates are operations in $\mathbb{F}$ and the wires are elements in $\mathbb{F}$.

2.2.2 Zero-Knowledge Proofs

To also make the aforementioned short proofs privacy-preserving, $\mathcal{P}$ can provide her own private input $\mathbb{w}$ to the computation, referred to a *witness*. Thus, Ψ now becomes a function of two inputs such as $\mathbb{y} = \Psi(\mathbb{x}, \mathbb{w})$. If $\mathcal{V}$ can be convinced that the statement $\mathbb{y} = \Psi(\mathbb{x}, \mathbb{w})$ is True without learning anything about $\mathbb{w}$, then the scheme is a ZKP protocol; such protocols become more powerful when the witness is a solution to an NP-hard problem. Most existing works leverage ACs where the algorithm is transformed to constraints and the proof convinces $\mathcal{V}$ that there exists a witness satisfying these constraints. Nevertheless, an important limitation of earlier works (e.g., [238]) is the need to know the target NP-hard algorithm beforehand, rendering them non-universal. In a simple, yet powerful example, $\mathcal{P}$ wants to prove that she knows a preimage (secret witness $\mathbb{w}$) for a hash digest chosen by $\mathcal{V}$, without revealing what the preimage is [119]. To formalize the above statement, let us assume an algorithm Ψ that implements a cryptographic hash function (e.g., SHA-256), compares the computed output to a public input $\mathbb{x}$ (i.e., the hash digest chosen by $\mathcal{V}$) and outputs a Boolean value $\mathbb{y}$ whether the two hashes match or not:

$$\mathbb{y} = \Psi(\mathbb{x}, \mathbb{w}) = \begin{cases} \mathsf{True}, & \text{if SHA-256}(\mathbb{w}) \equiv \mathbb{x} \\ \mathsf{False}, & \text{otherwise} \end{cases}$$

In a zero-knowledge protocol, $\mathcal{P}$ computes the value $\Psi(\mathbb{x}, \mathbb{w})$ locally without revealing $\mathbb{w}$, and provides strong cryptographic guarantees to $\mathcal{V}$ that Ψ was evaluated faithfully. If the result is $True$, then $\mathcal{V}$ is convinced that $\mathcal{P}$ knows a correct witness $\mathbb{w}$ to make SHA-256 return $\mathbb{x}$. The most prominent ZKP constructions rely either on *arithmetic circuits* [113,195] or on *random access machines (RAM)* [33,37]. The former class requires an expensive, trusted pre-processing phase that binds the two parties to a static arithmetic circuit for each different Ψ. Conversely, RAM-based protocols are more powerful and flexible as they do not depend on a specific computation and can verify any state transition in the RAM. Notably, we can also view the computation Ψ as a state machine $\mathbb{SM}$ executing a procedure as a sequence of state machine transitions.

2.2.3 Properties of Proof Systems

Every proof system should satisfy two basic properties:

- **Completeness.** If the statement $\mathbb{y} = \Psi(\mathbb{x})$ is True, an honest $\mathcal{P}$ should be able to convince an honest $\mathcal{V}$. In other words, given the same set of inputs, $\mathcal{V}$ should yield the same result $\mathbb{y}$ through the protocol as $\mathcal{P}$.

- **Soundness.** If the statement $\mathbb{y} = \Psi(\mathbb{x})$ is False, a malicious $\mathcal{P}$ cannot convince an honest $\mathcal{V}$ that it is True (except with negligible probability).

If the proof system also preserves the privacy of the prover's inputs, then it would satisfy the zero-knowledge property:

- **Privacy.** If the statement $\mathbb{y} = \Psi(\mathbb{x}, \mathbb{w})$ is true, $\mathcal{P}$ can convince $\mathcal{V}$ without leaking any information about $\mathbb{w}$.

In ZKP systems, we have two additional desired properties:

- **Transparency.** The proof system does not require any trusted setup (e.g., [31, 34, 61, 239]); any randomness used by transparent frameworks is public coins.

- **Scalability.** The proof system can gracefully handle programs and inputs of larger sizes, which makes it more practical. This property is applicable to both the prover and the verifier: The scalability of $\mathcal{P}$ corresponds to the overhead of generating the proof and convincing $\mathcal{V}$, and it should be somewhat similar to the time it would take to re-execute the target program. Likewise, the scalability of $\mathcal{V}$ entails that verification times are exponentially smaller than the cost of re-executing the target program (scalable verifiers are referred to as *succinct*).

A proof system that satisfies all the above properties is a *Zero-Knowledge Scalable Transparent ARgument of Knowledge (zk-STARK)* [31, 34].

2.2.4 A Primer on zk-STARKs

Overview. Typically, a ZKPK involves (1) a *witness* input $\mathbb{w}$ (i.e. the piece of data we want to prove knowledge for), and (2) verifiable execution of a public algorithm that tests an assertion about $\mathbb{w}$. For example, the latter can be an algorithm $\mathbb{A}$ that "multiplies two integers and compares the result with a composite N," whereas the witness can be a set of primes p, q. In ZKPK, we can prove knowledge of correct p and q if we can prove that $\mathbb{A}$ was executed faithfully on the witness input. Likewise, $\mathbb{A}$ can be a *modular Fibonacci loop* instantiated with integers a_0, a_1 so that each loop computes $a_i = a_{i-1} + a_{i-2} \bmod p$, where p is a large prime that defines a finite field. In this case, if $a_0 = 1$ and $a_1 = \mathbb{w}$ is our witness, we can prove knowledge of a suitable a_1 that returns the anticipated output $\mathbb{y}'$ after a large number of loops (i.e., $a_T = \mathbb{y}'$ at step T).

The zk-STARK methodology enables proving the integrity of the computation of algorithm $\mathbb{A}$ after T steps on input $\mathbb{w}$ that yields an output $\mathbb{y}$; this is possible using *arithmetization* and *low-degree testing* operations on polynomials over finite fields [31]. Specifically, arithmetization is the reduction of a computational problem (i.e., verifying a computation) into an algebraic problem, such as checking that a certain polynomial is of low degree. In zk-STARKs, arithmetization comprises three steps: (a) generating the execution trace of algorithm $\mathbb{A}$ for T steps, (b) generating a set of polynomials that express constraints for each execution step (e.g., if two values are multiplied, the result equals their product), and (c) combining the execution trace and polynomial constraints into a single polynomial $\mathbb{Q}$. Finally, the zk-STARK approach shows that it is possible to generate a ZKPK by employing error-correction methods (specifically Reed-Solomon proximity testing [31]) and show that the generated polynomial $\mathbb{Q}$ is actually low-degree. In this case, $\mathcal{V}$ is convinced that a polynomial is of low-degree (and thus the integrity of the computation of $\mathbb{A}$) with only a small number of queries to $\mathcal{P}$ [31].

Low-Degree Extension and Commitment. In the first step in a zk-STARK proof,

$\mathcal{P}$ encodes the execution trace of algorithm $\mathbb{A}$ into a sequence of states. In our earlier modular Fibonacci loop example, this would be the set $\{a_0, a_1, a_2, \ldots, a_T\}$ (note, $\mathcal{P}$ knows the correct witness a_1). $\mathcal{P}$ then generates a sequence $\{b_0, b_1, \ldots\}$ and pairs each state in the trace with the corresponding b_i (e.g., create (b_i, a_i) pairs). These pairs are interpreted as (x, y) points and are used to efficiently compute the *interpolating trace polynomial* $F(B)$ across them with the Lagrange interpolation method. Knowing the coefficients of the trace polynomial, $\mathcal{P}$ fixes an R so that $R \gg T$ computes its *low-degree extension (LDE)*, i.e., the values of $F(B)$ for $B = \{b_{T+1}, b_{T+2}, \ldots, b_R\}$. Finally, $\mathcal{P}$ computes a Merkle-tree over the sequence $\{F(b_0), F(b_1), \ldots, F(b_R)\}$ and commits to the tree root.

Arithmetization. zk-STARK employs a formal algebraic intermediate representation (AIR) of the target algorithm $\mathbb{A}$ as a set of low-degree polynomials $\{P_1, \ldots, P_K\}$ that encode K constraints about the execution trace of the computation [31]. In this case, the transition from step T to step $T+1$ in the computation is valid if and only if all constraints are satisfied (i.e., $P_1 = \cdots = P_K = 0$). In our Fibonacci example (with interpolated trace polynomial F), our constraints are $F(x+2) - F(x+1) - F(x) = 0$ for $x \in \{b_0, b_1, \ldots, b_{T-2}\}$, $F(x) - 1 = 0$ for $x = b_0$, and $F(x) - \mathtt{y}' = 0$ for $x = b_T$, where all operations are mod p.

By the *polynomial remainder theorem*, if b is a root of a polynomial $Q(x)$, then $Q(x) = (x-b)P(x)$, i.e., $P(x) = Q(x)/(x-b)$ is a polynomial. Therefore, we can can compute the AIR polynomials by factorizing the constraints of the computation. In our Fibonacci example we have $P_1(x) = (F(x) - 1)/(x - b_0)$, $P_2(x) = (F(x) - 1)/(x - b_T)$, and $P_3(x) = (F(x+2) - F(x+1) - F(x))/[\prod_{i=0}^{T-2}(x - b_i)]$. Moreover, zk-STARK computes the *Composition Polynomial (CP)* as the linear combination of all P_is and applies an LDE up to R points.

Low-degree testing. The goal of this step is to convince $\mathcal{V}$ that the *distance* between the computed CP and a low degree polynomial is relatively small, where the distance between any function and a polynomial of degree d is defined as the number of x

inputs where their values are different. In zk-STARK, this is possible using the *FRI operation* (Fast Reed-Solomon Interactive Oracle Proofs of Proximity), which reduces the problem of proving that a function of domain size R is close to polynomial of degree bounded by d into a new smaller problem where the function domain size is $R/2$ and the polynomial degree is $d/2$ [31]. FRI is applied iteratively to CP (treated as a function of domain size R after LDE) until $d = 1$; each iteration replaces every polynomial power x^i with $x^{\lfloor i/2 \rfloor}$ and the coefficients of same powers are added. Also, after each FRI iteration, $\mathcal{P}$ computes a Merkle tree of the values of CP over its entire domain and the tree root is committed.

Decommitment Step. In this step, $\mathcal{V}$ is convinced that the original execution trace for algorithm $\mathbb{A}$ was computed faithfully by sending queries to $\mathcal{P}$. Specifically, the verifier selects a small set of b_i values for $i \in \{0, 1, \ldots, R\}$ and for each one the prover reveals the Merkle-tree path for her commitments for each FRI step and the LDE of the trace polynomial F. $\mathcal{V}$ uses the values of F and reconstructs the corresponding CP values (for each FRI step). If all commitments are correct, $\mathcal{V}$ is convinced the proof is sound with very high probability [31].

2.2.5 Fiat-Shamir Heuristic

Fiat and Shamir introduced a method (known as the *Fiat-Shamir heuristic*) that eliminates the interaction in public coin proof-of-knowledge protocols and make them non-interactive [106]. Essentially, the Fiat-Shamir heuristic relies on the random oracle model [28] in order to substitute the random challenge generated by the verifier with a challenge that is the output of a hash function over the previous protocol messages. The sequence of messages (and most importantly their commitments) in the protocol are referred to as a *transcript*. Modeling the hash function as a random oracle ensures the unpredictability of the verifier, and public coins are obtained from the hash digest bits using the transcript as input. However, this construction is secure as long as the hash function is indeed a random oracle that prevents the prover from guessing its output before the commitments are generated.

Algorithm 1 Lightweight Collision-Resistant Hash Function

Input: $k \in \mathbb{Z}_q$, $k' \in \mathbb{F}_{2^n}$, $m = (m_1, m_2, \ldots, m_\ell) \in \mathbb{Z}_q^\ell$

procedure $LCRHF_{k,k'}(m)$
  $\mathcal{H} \leftarrow m_1 \cdot k \bmod q$ ▷ q is a prime
  for $i \leftarrow 2$ to ℓ **do** ▷ Horner's method
    $\mathcal{H} \leftarrow k \cdot (\mathcal{H} + m_i) \bmod q$ ▷ $\ell - 1$ iterations
  return $\textsc{SpeckEncrypt}_{k'}(\mathcal{H})$ ▷ Speck alg. [21]

2.3 Lightweight Pseudorandom Functions with Extended Input

A deterministic function that can be efficiently computed on an input block and generate an output block that is computationally indistinguishable from an output generated by a truly random function is called a pseudorandom function (PRF) [168]. In practice, PRFs can be instantiated using secure block ciphers (such as Speck [21]), which combine a secret key with an input plaintext block to transform the plaintext into a (randomly looking) ciphertext block. Leveraging a universal hash function (UHF), the fixed input block size of a PRF can be extended to a secure PRF of arbitrary input size by applying a UHF operation before invoking the PRF (i.e., a $PRF(UHF(\cdot))$ composition) [217, Section 4.2].

Based on the blueprint of Carter and Wegman [67], a UHF $\mathcal{U}$ can be constructed as a polynomial of degree-ℓ modulo a prime number q that is evaluated at input point k. As discussed in [230], $\mathcal{U}$ can be defined over ℓ input blocks m_1 to m_ℓ (treated as polynomial coefficients) and a secret key k. In this case, $\mathcal{U}_k(m_1, \ldots, m_\ell) = m_1 k^\ell + m_2 k^{\ell-1} + \cdots + m_\ell k \bmod q$, with $m_i \in \mathbb{Z}_q$, is a lightweight UHF with a collision probability $\varepsilon \leq \ell/q$ for any pair of distinct inputs [158]. Using Horner's method, $\mathcal{U}$ can be computed iteratively and can be encrypted using the Speck cipher with key k' to construct an efficient, secure PRF with input extended over ℓ blocks. This PRF prevents adversaries from detecting if a collision has occurred and is secure against forgery attacks [147]. Alg. 1 computes the hash digest of message m with keys k, k'.

2.4 Paillier Cryptosystem

Homomorphic encryption allows performing certain mathematical operations on encrypted data that correspond to applying related operations on unencrypted data,

without performing any decryptions. The Paillier partially homomorphic encryption (PHE) scheme supports homomorphic addition of encrypted messages [194]. Paillier uses a public/private key pair and supports probabilistic encryption (paired with deterministic decryption); its additive homomorphic property can be summarized as follows: Given two ciphertexts $c_1 = \mathsf{Enc}_{pk}(m_1)$ and $c_2 = \mathsf{Enc}_{pk}(m_2)$, one can compute $c = \mathsf{Enc}_{pk}(m_1 + m_2)$ without having to decrypt c_1 or c_2. (Here we omit the randomness for clarity.) Below we outline the basic operations of the Paillier cryptosystem.

Key setup. Let N be the security parameter defined as the product of two large primes p and q, such that $\mathsf{GCD}(pq, (p-1)(q-1)) = 1$, and $\lambda = \mathsf{LCM}(p-1, q-1)$. Select a random generator g in $\mathbb{Z}^{\times}_{N^2}$ and ensure that N divides the order of g by checking whether $\mu = (L(g^{\lambda} \bmod N^2))^{-1} \bmod N$ exists, where L is defined as $L(u) = {}^{(u-1)}/_{N}$. The public key is $pk = (N, g)$ and the secret key is $sk = (\lambda, \mu)$.

Encryption. Randomly select a value ρ from $\mathbb{Z}^{\times}_{N}$ so that $\mathsf{GCD}(\rho, N) = 1$. Paillier encryption is defined as a unique correspondence between ρ and plaintext $m \in \mathbb{Z}_N$ with a value $c \in \mathbb{Z}^{\times}_{N^2}$ so that $c = \mathsf{Enc}_{pk}(m, \rho) = g^m \cdot \rho^N \bmod N^2$.

Decryption. The decryption algorithm performs an inverse mapping from $\mathbb{Z}^{\times}_{N^2}$ to $\mathbb{Z}_N$ and is defined as $m = \mathsf{Dec}_{sk}(c) = (L(c^{\lambda} \bmod N^2) \cdot \mu) \bmod N$.

Homomorphic operations. The Paillier cryptosystem has a notable homomorphic property that enables the product of two ciphertexts to decrypt to the sum of their corresponding plaintexts, based on the following observation:

$$\begin{aligned} &\mathsf{Enc}_{pk}(m_1, \rho_1) \cdot \mathsf{Enc}_{pk}(m_2, \rho_2) \\ &= \left(g^{m_1} \cdot {\rho_1}^N\right) \cdot \left(g^{m_2} \cdot {\rho_2}^N\right) \bmod N^2 \\ &= g^{m_1+m_2} \cdot (\rho_1 \cdot \rho_2)^N \bmod N^2 \\ &= \mathsf{Enc}_{pk}(m_1 + m_2, \rho_1 \cdot \rho_2). \end{aligned} \tag{2.1}$$

In general, the product of P ciphertexts is equivalent to the encryption of the summation of the P corresponding plaintexts (again, we omit the randomness ρ):

$$\prod_{i=1}^{P} \mathsf{Enc}_{pk}(m_i) = \mathsf{Enc}_{pk}\left(\sum_{i=1}^{P} m_i\right) \bmod N^2. \tag{2.2}$$

Finally, the Paillier cryptosystem allows multiplication of a ciphertext with a constant, which results in the encryption of the product of the plaintext with the constant, as follows:

$$\begin{aligned} \mathsf{Enc}_{pk}(m_1, \rho)^{m_2} &= \left(g^{m_1} \cdot \rho^N\right)^{m_2} \bmod N^2 \\ &= g^{m_1 \cdot m_2} \cdot \rho^{N \cdot m_2} \bmod N^2 = \mathsf{Enc}_{pk}(m_1 \cdot m_2, \rho^{m_2}). \end{aligned} \tag{2.3}$$

2.5 Commitment Schemes

A secure commitment scheme is a cryptographic primitive that uses an algorithm Com to enable one party, called a sender $\mathcal{S}$, to commit to a value x while keeping it *hidden*. To do so, $\mathcal{S}$ publishes a value $c = \mathsf{Com}(x, r)$ that was generated using randomness r; later, $\mathcal{S}$ can "open" the commitment by disclosing the used values x, r. The algorithm Com *binds* $\mathcal{S}$ to the hidden value x and offers provable guarantees to a receiving party $\mathcal{R}$ that the value revealed later was actually the same as the one $\mathcal{S}$ originally committed to when c was first published. Early commitment schemes proved that it is possible for example to have two parties play a card game or flip a coin via the telephone without having to trust each other [43, 58, 213].

For any commitment scheme to be secure, it must satisfy two basic properties: *binding* and *hiding*. The first property guarantees that upon committing to a secret value x, $\mathcal{S}$ cannot change x later. Formally, for all non-uniform probabilistic polynomial-time algorithms that output (x_1, r_1) and (x_2, r_2), the probability that $\mathsf{Com}(x_1, r_1) = \mathsf{Com}(x_2, r_2)$ with $x_1 \neq x_2$, is negligible. The second property requires that the commitment c should not reveal any information about the value x before opening, that is, for all non-uniform probabilistic polynomial-time algorithms, the probability of extracting any information about x from c should be negligible. A more recent commitment scheme, named after T. P. Pedersen, features an additive homomorphic property so that for messages x_1 and x_2 with blinding factors r_1 and r_2 we have: $\mathsf{Com}(x_1, r_1) \cdot \mathsf{Com}(x_2, r_2) = \mathsf{Com}(x_1 + x_2, r_1 + r_2)$ [197].

2.6 Secret Sharing

Secret sharing schemes allow a dealer to distribute shares of her data to multiple parties so that each share does not reveal anything about the original data [25]. Secret sharing allows any sufficient subset of parties to reconstruct the secret by combining their shares and at the same time, any smaller subset of parties cannot reveal any partial information about the secret. In MPC, each party creates secret shares of their data and shares them with the other parties. MPC utilizes secret sharing to compute arbitrary arithmetic functions as arithmetic circuits [25, 87, 148, 211]. After the computation is done, the parties combine the shares to reconstruct the final output.

2.7 Distributed Point Functions (DPF)

Function secret sharing (FSS) [55] enables splitting the output of a function f into additive shares, where each share of the function is represented by a separate key. Each key allows the owner to efficiently generate an additive share of the output $f(x)$ on a given input x. DPFs are a special case of FSS where f is a point function $f_{\alpha,\beta}(x) := \beta$ if $x = \alpha$, or 0 otherwise. A DPF consists of two algorithms: Gen and Eval. The Gen algorithm takes as input the function $f_{\alpha,\beta}$ and outputs two keys key_0 and key_1. The Eval algorithm evaluates an input x such that $\mathsf{Eval}(0, \mathsf{key}_0, x) + \mathsf{Eval}(1, \mathsf{key}_1, x) = \beta$ for $x = \alpha$, and 0 for $x \neq \alpha$. Privacy ensures (α, β) remains hidden from an adversary in possession of one of the keys (but not both). We discuss DPFs and other stronger notions, such as incremental DPFs (IDPF) [49] and verifiable DPFs (VDPF) [94].

2.7.1 Incremental DPF (IDPF)

An IDPF [49] generalizes the DPF notion by secret-sharing an "incremental point function", i.e., the "point" in DPF is now a path on a full binary tree from the root to one of the leaves. Here we take α value is a bit string and $f_{\alpha,\beta}(x) = \beta$ if x is a prefix of α and 0 otherwise. Although using multiple DPFs could achieve the task of having a path from the root to the leaves, IDPFs perform this task with linear cost

in the number of bits n for strings that share common prefixes [49]. Using standard DPFs this cost would grow to $\mathcal{O}(n^2)$.

Similarly to DPFs, an IDPF has two main operations, a key-generation algorithm, and an evaluation algorithm. The key-generation algorithm takes as input the function $f_{\alpha,\beta}$ and outputs two keys key_0 and key_1. The evaluation algorithm takes as input a candidate prefix string x and a key and returns a share of $f_{\alpha,\beta}(x)$.

2.7.2 Verifiable DPF (VDPF)

The work of [94] considers efficient hashing-based verifiable properties to ensure that a DPF key is well-formed. Moreover, VDPFs enable a batched verification procedure with communication proportional to the security parameter. However, the verifiability property of VDPFs works only for DPF and not for IDPF. We present the VDPF algorithms below:

- VDPF.$\mathsf{Gen}(1^\kappa, f_{\alpha,\beta}) \rightarrow (\mathsf{key}_0, \mathsf{key}_1)$. Given the security parameter 1^κ and a function f, output keys $\mathsf{key}_0, \mathsf{key}_1$.

- VDPF.$\mathsf{BatchEval}(b, \mathsf{key}_b, \mathbf{X}) \rightarrow (\mathbf{Y}_b, \pi_b)$: For $b \in \{0,1\}$, batch verifiable evaluation takes a set $\mathbf{X} \coloneqq \{x_1, x_2, \ldots, x_m\}$, where each $x_i \in \{0,1\}^n$. It outputs $\mathbf{Y}_b \coloneqq \{y_{b,1}, y_{b,2}, \ldots, y_{b,m}\}$.

Correctness ensures that $\mathbf{Y}_0 + \mathbf{Y}_1 = f_{\alpha,\beta}(\mathbf{X})$. Privacy ensures that an adversary in possession of one of the keys (but not both) does not obtain any information about the function f. The verifiability property of VDPF ensures that the proofs π_0 and π_1 are the same if and only if they have been generated from valid keys key_0 and key_1 of a point function.

Chapter 3

ZILCH: A FRAMEWORK FOR DEPLOYING TRANSPARENT ZERO-KNOWLEDGE PROOFS

As cloud computing becomes more popular, research has focused on usable solutions to the problem of verifiable computation (VC), where a computationally weak device (Verifier) outsources a program execution to a powerful server (Prover) and receives guarantees that the execution was performed faithfully. A Prover can further demonstrate knowledge of a secret input that causes the Verifier's program to satisfy certain assertions, without ever revealing which input was used. State-of-the-art *Zero-Knowledge Proofs of Knowledge* (ZKPK) methods encode a computation using arithmetic circuits and preserve the privacy of Prover's inputs while attesting to the integrity of program execution. Nevertheless, developing, debugging, and optimizing programs as circuits remains a daunting task, as most users are unfamiliar with this programming paradigm.

In this work, we present Zilch, a framework that accelerates and simplifies the deployment of VC and ZKPK for any application *transparently*, i.e., without the need for a trusted setup. Zilch uses traditional instruction sequences rather than static arithmetic circuits that would need to be regenerated for each different computation. Towards that end, we have implemented zMIPS: a MIPS-like processor model that allows verifying each instruction independently and composing a proof for the execution of the target application. To foster usability, Zilch incorporates a novel cross-compiler from an object-oriented Java-like language tailored to ZKPK and optimizes our zMIPS model, as well as a powerful API that enables integration of ZKPK within existing C/C++ programs. In our experiments, we demonstrate the flexibility of Zilch using

two real-life applications and evaluate Prover and Verifier performance on a variety of benchmarks.

3.1 Introduction

Cloud computing offers on-demand computational power, emerging as an ideal solution for outsourcing computation from relatively weak devices (phones, laptops, IoT). However, delegating computation to an untrusted third-party running unreliable software, and potentially untested – or malicious [227, 228] – hardware, comes with many risks [223]. Data may even be corrupted while at rest [170], or in transit. Moreover, cloud service providers may have monetary incentives to either skip computation steps (e.g., skip computing all decimal points on a big number) or completely counterfeit a result. *How can we trust the results computed by the cloud and be assured that the computation was carried out faithfully?*

Verifiable Computation (VC) leverages mathematical and cryptographic primitives, such as probabilistically checkable proofs (PCPs) [11, 12], interactive proofs [15, 123, 124, 169, 212] and commitment-based argument schemes [41, 58, 112, 113, 151, 175], to provide strong guarantees to a client on the correct evaluation of a statement in NP. In these schemes, one party $\mathcal{P}$ (the prover) generates and commits to a proof that the computation was executed faithfully and another party $\mathcal{V}$ (the verifier) performs unpredictable tests to efficiently check the integrity of the execution. Given an honest $\mathcal{P}$, these tests can convince $\mathcal{V}$. Conversely, a faulty execution would be noticed by the verifier with a very high probability. Intuitively, in such protocols the overhead for the prover and the complexity of performing the tests for $\mathcal{V}$ should be less than the whole execution on the computationally-weak device; i.e., the verifiable outsourcing is practical. Numerous systems [236–238] strive to bring verifiable outsourcing one step closer to practicality.

A notable extension to verifiable computation is to enable the prover to apply the computation on a secret input – also called the *witness* – which is never revealed to $\mathcal{V}$. This approach, known as a *zero-knowledge proof* (ZKP), allows the prover to

convince the verifier that she knows a secret without actually disclosing it [33, 37, 84, 195]. For instance, ZKPs can be leveraged to log in to a website without typing a password by simply sending a proof that you "know the valid password". ZKP-based authentication eliminates the need for maintaining server-side password databases, sending passwords via unsafe channels, having to digitally sign challenge messages that could later be misused, or even having to disclose an intellectual property for functional verification [182].

Although ZKPs and VC have numerous applications, many state-of-the-art solutions (e.g., [33, 37, 195, 222, 236]) suffer from a severe limitation: they require a trusted authority to generate the public parameters for the system and then eliminate any knowledge of the randomness used to generate them (referred to as *toxic waste*). A malicious third party that obtains access to that toxic waste can forge false proofs and trick an honest verifier. Having a single point of failure in VC and ZKP systems that rely on cryptographic primitives seems contradictory. As a result, systems that use public randomness and thus have a *transparent setup* have been proposed [7, 31, 34, 61, 156, 239, 249]. Likewise, the works in [62, 173] provide constructions leveraging updatable common reference strings (CRS).

Another observation about the computational model followed by most VC and ZKP systems is the need to express computer programs as arithmetic circuits, or equivalently as a set of arithmetic constraints over a finite field $\mathbb{F}$. Works such as [84, 195, 238] provide a compiler from a high-level language (typically a subset of `C`) to arithmetic circuits, however, they require the circuit to be fixed offline during the trusted setup phase. Notably, this conversion is laborious as it is hard to express arbitrary algorithms using arithmetic circuits and even harder to edit, debug, or optimize these VC circuits without deep knowledge of cryptography and circuit design. Furthermore, these circuits are program-specific and cannot be reused to verify other programs, which renders them non-universal. An alternative is to employ a Random Access Machine model that provides a low-level language, such as TinyRAM [37], that can define a universal circuit. However, program development using such esoteric machine models without high-level

toolchain support still requires significant effort from a programmer's perspective.

Unfortunately, neither of these models of computation is natural for most non-crypto-savvy programmers. Thus, an important objective of this work is to develop a methodology for verifiable computation and zero-knowledge proofs of knowledge using a convenient programming model that does not rely on any trusted third party. Our approach is to leverage a programming model that is based on a *sequence of instructions* instead of a circuit netlist. At the same, an additional goal is to optimize performance while ensuring programming convenience.

The main contribution of this work is the development of Zilch[1], a specialized framework to facilitate the development of interactive zero-knowledge proofs for any application. Zilch enables the development of algorithms used for VC and ZKPs using our high-level language called ZeroJava, which is compiled into an intermediate representation (IR) that is ultimately transformed into a set of mathematical constraints. ZeroJava is an intricately chosen subset of Java specifically tailored for deploying zero-knowledge arguments, while the IR statements are evaluated in our custom abstract machine (called zMIPS) that is adopted from a MIPS processor. To generate and verify proofs, Zilch leverages the state-of-the-art zk-STARK library [31] that does not rely on any trusted third-party setup (i.e., contrary to other libraries [33, 37, 222]). Moreover, zk-STARK is resilient against attacks by large-scale quantum computers[2] and its security relies on collision-resistant hash functions [151] and the random oracle model [106]. In addition to newly developed applications that can be developed in ZeroJava, our Zilch framework offers a powerful API that is compatible with C/C++ programs to facilitate embedding VC and ZKPs into existing code. In all cases, Zilch enables a prover to interact with a verifier automatically over a network.

Contributions. Our contributions are summarized as follows:

[1] **Zilch** / zɪltʃ / : zero; nothing. *The search came up with zilch.*

[2] We refer to a system's property of being resilient to known attacks by large-scale quantum computers as *plausible post-quantum secure*.

- Design of zMIPS, an abstract machine that is adopted from the MIPS processor with judiciously selected instructions that create arithmetic circuits and enable zero-knowledge proofs for any target application,
- Design and implementation of the ZeroJava high-level language, with a cross-compiler from ZeroJava to zMIPS and an assembly optimizer,
- Development of the Zilch framework and API to facilitate the construction of ZKPs for existing C/C++ applications.

3.2 The Zilch Framework

3.2.1 Our Threat Model

Cheating Prover. To mitigate the risks applicable to our approach for computational integrity, our threat model assumes an adversary given access to the prover's capabilities. The adversary succeeds if she produces a false statement that will convince $\mathcal{V}$ to accept it. In the VC scenario, the cheating $\mathcal{P}$ has incentives to skip some steps or completely forge the result, while in the ZKPK case, the adversary tries to convince $\mathcal{V}$ that she knows the witness without actually knowing it. Zilch features a configurable security parameter λ, which determines the probability ($\leq 2^{-\lambda}$) that an adversary can successfully deceive an honest verifier in the above experiments. Thus, λ defines the soundness property of the proof system.

Cheating Verifier. On the other hand, a *malicious* verifier is assumed to behave without restrictions and not necessarily follow the protocol specification in order to extract any information about the secret input $\mathbb{w}$. If $\mathcal{P}$ follows the protocol correctly and the statement is true, $\mathcal{V}$ will never learn any (private) witness data from the interaction with the prover except the fact that the statement is true, (i.e., zero-knowledge property). Moreover, we don't consider trivial cases where $\mathcal{V}$ completely withdraws from the protocol of the two parties.

Post-quantum resilience. Many different VC and ZKP frameworks rely on elliptic curves and pairings [33, 36, 37, 112, 126, 129, 195, 210, 222], as well as the discrete

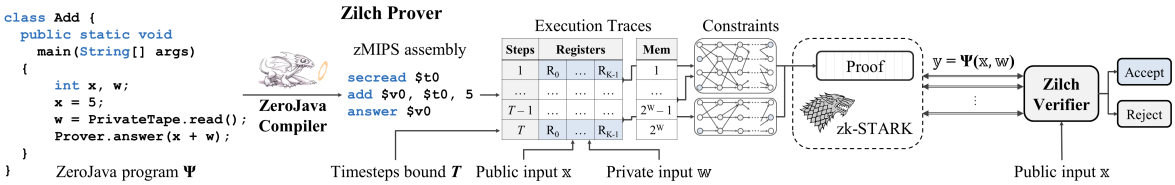


Figure 3.1: Zilch Framework Overview. Using our ZeroJava compiler, $\mathcal{P}$ provides the assembly code to Zilch along with the private and public inputs. Zilch first determines the steps bound T automatically and then computes the result $\mathbb{y}$. Finally, $\mathcal{P}$ and $\mathcal{V}$ interact over a limited number of rounds and the verifier either accepts the proof (i.e., she is convinced) or rejects it.

logarithm problem [51, 61]; this makes them susceptible to attacks using quantum computers [39, 199, 215, 216]. Aurora [34], Ligero [7], and zk-STARKs [31] rely on collision-resistant hash functions, which renders them resilient to known attacks from quantum computers. Likewise, since Zilch utilizes zk-STARK as its back-end proof system, it inherits the same post-quantum resilience property.

Transparency. Previous VC and ZKP solutions (e.g., [33,37,129,195,222,238]) require a third party to set-up the system with non-public randomness. If that party is not trustworthy, this secret randomness could be misused to generate false proofs and compromise the system's security. Zilch, like other state-of-the-art PCP-based systems (e.g., [7, 31, 239]) is transparent as it relies *only on public randomness* and does not need any trusted third party during its set-up phase.

3.2.2 Key Observations in our Methodology

Zilch aims to facilitate proving computational integrity statements; in particular, our goal is to convince a verifier that a computer program implemented as a sequence of well-defined instructions returns an expected output for a set of inputs. For our methodology, we observe that in order to verify the execution integrity of an algorithm, it is sufficient to *divide it into two parts, verify both individually and finally verify a correct transition from the first part to the second.* This observation can be applied in a divide-and-conquer manner to recursively decompose any algorithm into sub-algorithms until each becomes simple enough to be verified individually. Each

individual proof is then combined into a composable proof for the execution of the original algorithm.

A second important observation about computational integrity in our case is that it is sufficient to decompose the target algorithm up to *the granularity of individual assembly instructions* and prove the integrity of each instruction directly using its corresponding arithmetic circuit (AC). This offers great flexibility, as a predetermined set of assembly instructions (each mapped to a small AC) can be combined to define any arbitrary algorithm. Conversely, trying to verify any large program directly using a large static AC would require generating a unique AC for each different program, which is exactly the daunting task that we are trying to avoid in the first place. Notably, our proposed method of verifying programs at the granularity of an assembly instruction is beneficial, as it is relatively easy to translate any program written in a high-level language into a set of assembly instructions using a compiler. To prove the integrity of execution, we first verify the AC of each individual assembly instruction and finally verify each state transition between consecutive instructions.

3.2.3 Overview of our Framework

To instantiate our methodology, we have developed Zilch: a transparent and post-quantum resilient programming framework for creating ZKPK for any application. Zilch is universal since it takes as input *a description of a Turing Machine (TM)* (i.e., a computer program) and *two input tapes*, one private and one public. More formally, Zilch implements a time-bounded Universal TM and can be used for any arbitrary computation that is expressed as a sequence of instructions. Internally, Zilch adopts a MIPS-like processor model (i.e., an abstract machine with memory, program counter, registers, and fetch-decode-execution pipeline stages) called zMIPS; our machine supports a judiciously selected instruction set that can implement and verify any computation in zero-knowledge.

Zilch consists of a front-end and a back-end. The front-end defines our customized subset of Java specifically tailored to zero-knowledge arguments, called ZeroJava, and includes our compiler for translating the ZeroJava high-level code into zMIPS assembly instructions. From a programmer's perspective, ZeroJava is *object-oriented and strongly-typed* like Java, while excluding Java features that complicate the run-time system, such as exceptions and multi-threading. Our compiler comprises four phases: (a) transforming the high-level code into an intermediate representation (IR), (b) performing static analysis on the IR to optimize it, (c) performing register allocation to minimize the number of required registers, and (d) generating the zMIPS assembly. We elaborate more on the design choices of the Zilch front-end (i.e., the ZeroJava language and the compiler) in Section 3.2.4.

The Zilch back-end defines the zMIPS abstract machine that consumes zMIPS instructions to transition from one state to another; *each state comprises the program counter, K registers, and memory.* A computation is expressed as a sequence of instructions or equivalently as a sequence of abstract machine states. Furthermore, each assembly instruction generates individual constraints that must hold between each two consecutive abstract machine states, and having a finite set of instructions renders verification feasible. The sequence of states $\{S_1, S_2, ..., S_T\}$ forms an execution trace of a program (also called a *transcript*) that is T steps long; in each step, an instruction is fetched, decoded and executed by our abstract machine. The transcript can be represented as a two-dimensional table with T rows and K columns (Fig. 3.1), where each row represents a single execution step and each column tracks one zMIPS register through time. Two states S_i and S_{i+1} are valid if the machine in state S_i can transition with some instruction to state S_{i+1} in the next step. Depending on the instruction that operates on S_i, the new S_{i+1} state is different, since each instruction performs a unique transition from S_i to S_{i+1}. Given a time bound T, an execution trace tr of a specific program Ψ is valid if there exist public and private inputs $\mathbb{x}$, $\mathbb{w}$, such that the generated trace of Ψ on inputs $\mathbb{x}$ and $\mathbb{w}$ is tr. Likewise, the integrity of the memory state is ensured using a memory trace as will be discussed in Section 3.2.5. For each zMIPS

instruction, our Zilch back-end invokes the corresponding AIR constraints employing the zk-STARK library [31] (described in Section 2.2.4); using this library, Zilch consumes the transcript and the constraints, generates a low-degree polynomial, and then $\mathcal{P}$ is able to convince $\mathcal{V}$ that all polynomial constraints are satisfied in the execution trace for a secret witness $\mathbb{w}$.

Benefits of Zilch. Expressing a computation as a transcript of state transitions enables our abstract machine to generate universal ACs that do not require a different setup each time a new program is executed since each instruction implements its own AC. The only requirement is to provide an upper bound for the time steps of the target program in order to generate an AC that simulates the entire execution. Zilch can automatically determine the minimum number of execution time steps required for a specific program Ψ to generate a result by first simulating the computation quickly (without generating any constraints or proofs), and then checking if the output matches the expected result $\mathbb{y}$. If not, Zilch doubles the time steps bound T and repeats the same check with the new bound. If the computation returns $\mathbb{y}$ using the new time steps bound, Zilch generates the AIR constraints for the identified bound T and interacts with the verifier.

The zk-STARK library enables the development of interactive proofs where the prover and the verifier communicate over several rounds until the latter is convinced of the correctness of a proof. As we illustrate in Fig. 3.2, $\mathcal{P}$ and $\mathcal{V}$ initially agree on the polynomial constraints for the computation and then the prover generates the transcript and sends its encoding to the verifier. As soon as the verifier confirms the consistency of the encoding, the two parties interact over several rounds; in each round $\mathcal{V}$ first sends a message to $\mathcal{P}$ comprising *public random coin-flips*, and then $\mathcal{P}$ replies with *an oracle* (i.e., a long message comprising a proof) that the verifier can query probabilistically at any index of her choice. As Zilch employs the zk-STARK protocol for verification, we realize this oracle using Merkle tree commitments, and the corresponding cost for $\mathcal{V}$ is poly-logarithmic in the time steps required to execute

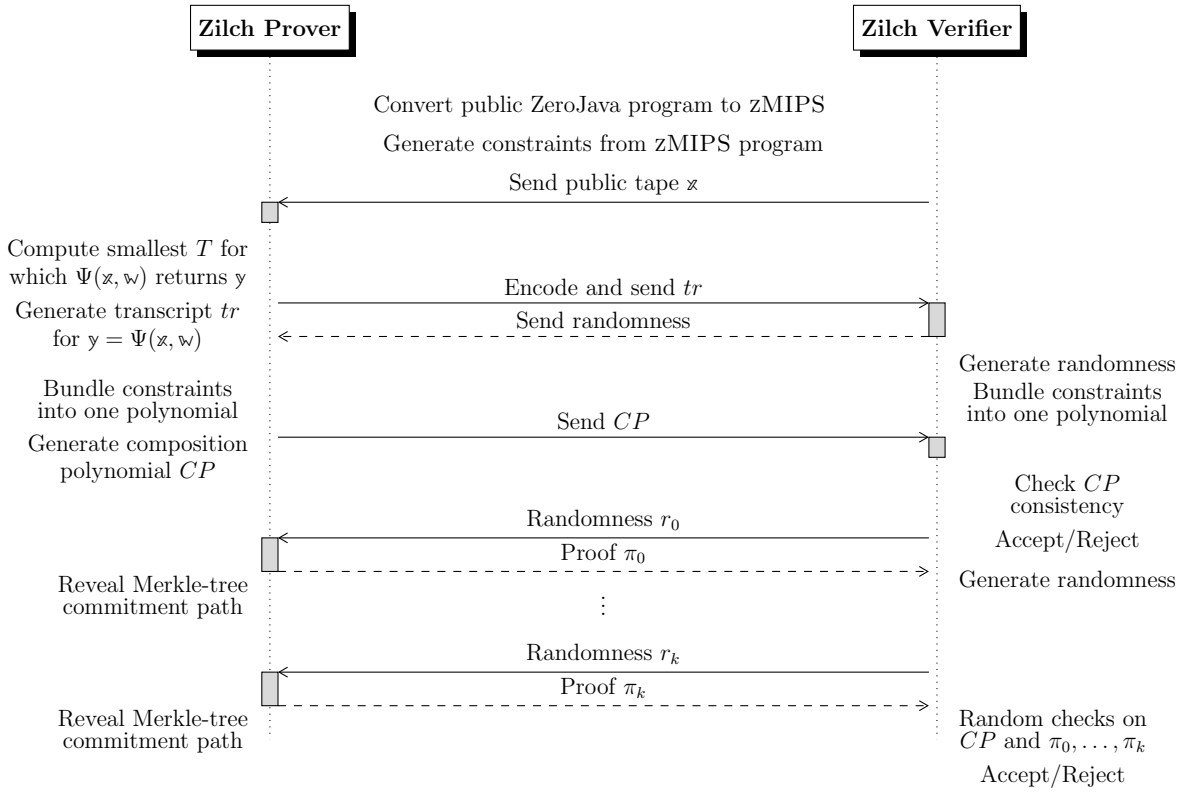


Figure 3.2: Prover and Verifier Interaction. Starting from a public computation expressed in ZeroJava and a public tape, $\mathcal{P}$ and $\mathcal{V}$ agree on the polynomial constraints. Next, $\mathcal{P}$ generates and encodes the transcript tr, and combines it with the polynomial constraints into a single composition polynomial CP that is shared with $\mathcal{V}$. Finally, the two parties engage in the FRI protocol and $\mathcal{V}$ either accepts or rejects $\mathcal{P}$'s statement.

Ψ, which renders our framework *succinct*.[3] Another important benefit of Zilch is its ability to serialize all the rounds of interaction between $\mathcal{P}$ and $\mathcal{V}$ that are shown in Fig. 3.2, which enables zero-knowledge proof verification over a network. In Section 3.4, we present how the communication cost scales with an increasing number of zMIPS instructions.

An overview of the Zilch framework is presented in Fig. 3.1, where a simple ZeroJava program for addition is translated into zMIPS assembly instructions using our compiler. Zilch then produces an execution trace that is ultimately transformed into AIR constraints for the zk-STARK library, and $\mathcal{P}$ can interact with $\mathcal{V}$. For an

[3] *Succinctness* denotes short proofs and scalable verification time.

honest prover, the verifier is convinced (i.e., accepts the proof) that the program was executed faithfully and that $\mathbb{y} = \Psi(\mathbb{x}, \mathbb{w})$ after at most T steps. Conversely, if the prover is malicious, the verifier will reject the proof with a very high probability.

3.2.4 Zilch Front-End Design

ZeroJava Language. To facilitate the development of ZKPK for any application, we introduce a self-contained high-level language called *ZeroJava*, which enables implementing arbitrary NP statements and is specifically tailored to VC and ZKPK. Contrary to previous approaches on high-level languages for verifiable computation (e.g., [84, 195, 238]), ZeroJava supports dynamic loop conditions without the need for unrolling (i.e., mutable state and iteration, dynamic termination, and infinite loops are supported). Moreover, ZeroJava supports ZKP-specific built-in functions that invoke specific zMIPS assembly instructions (we elaborate on these methods in Section 3.2.5 that discusses the zMIPS ISA). The following paragraphs present our design choices for ZeroJava.

ZeroJava is object-oriented and strongly typed, like Java. The basic types of ZeroJava are `int` for configurable W-bit signed integers (e.g., 32 bits), `boolean` for logical values, and `int[]` for arrays of integers. Integers are represented using two's complement, and overflows wrap around as in standard Java. Classes contain attributes and methods with arguments and return types of basic or class types. ZeroJava supports single inheritance without interfaces and function overloading (i.e., each method name must be unique). In addition, all methods are inherently polymorphic so that a method can be defined in a subclass if it has the same return type and arguments as in the parent. Fields in the base and derived class are allowed to have the same names and are essentially different fields. All ZeroJava methods are public and all fields are protected so that a class method cannot access fields of another class, with the exception of its parent; a class's own methods can be called via `this`. Local variables can be defined at the beginning of a method and can shadow the fields of the surrounding class with the same name.

Table 3.1: ZeroJava Language Operators

Assignment	Increment & Decrement	Arithmetic & Bitwise	Logical & Relational
`a = b`	`a++`	`a + b`	`!a`
`a += b`	`a--`	`a - b`	`a && b`
`a -= b`	**Arrays**	`a * b`	`a || b`
`a *= b`	`a[b]`	`a / b`	`a == b`
`a /= b`	`a.length`	`a % b`	`a != b`
`a %= b`	`new int[a]`	`a ^ b`	`a < b`
`a ^= b`	**Ternary**	`a & b`	`a > b`
`a &= b`	`(a) ? b : c`	`a | b`	`a <= b`
`a |= b`		`~a`	`a >= b`
`a <<= b`		`a << b`	
`a >>= b`		`a >> b`	

In ZeroJava, the `new` operator calls a default void constructor. In addition, there are no inner classes and there are no static methods or fields. A ZeroJava program begins with a special main class that does not have fields and methods and contains the main method (i.e., `public static void main(String[] args)`). After the main class, other classes may be defined that can have fields and methods. In Table 3.1 we summarize all the ZeroJava supported operators.

Tapes. ZeroJava supports both public and private inputs via two read-only input files called *tapes*. Each tape can be read sequentially using the `READ` built-in method (the next word is consumed), or with the random access `SEEK` function (the word at a given offset is read). These built-in methods have an one-to-one correspondence with zMIPS instructions. In the case of ZKPK, the secret input (witness `w`) should be provided in the private tape; for VC, only the public tape is required.

ZeroJava Example. In Fig. 3.3, we provide a ZeroJava program that implements Wegner's efficient algorithm [242] to compute the Hamming weight of a secret number and then compare it with a public threshold. This example highlights various features of ZeroJava, such as reading from the public and the private tape, performing loops, as well as applying arithmetic, logical, and bitwise operations. Besides, a potential

```
class HammingWeightThreshold {
  public static void main(String[] a) {
    int threshold = PublicTape.read();
    int num = PrivateTape.read();
    int count = 0;
    while (num > 0) {
      num &= num - 1;
      count++;
    }
    Prover.answer(count > threshold);
  }
}
```

Figure 3.3: ZeroJava program to prove that a secret number has a Hamming weight that is greater than a public threshold.

application of this algorithm could be to compute the Hamming weight of a private RSA exponent in zero-knowledge and convince a verifier that it is greater than a threshold; this could offer additional assurance against certain attacks (e.g., the authors of [111] demonstrate a birthday attack on RSA private exponents with low Hamming weight).

ZeroJava Compiler. Using our ZeroJava compiler, programmers can translate NP statements expressed in ZeroJava high-level code into optimized zMIPS machine code. Since ZeroJava is a strongly typed language, the first step performed by our compiler is to statically analyze the program and verify its type safety, i.e., ensure that the types of expressions are consistent. For instance, a variable declared as an integer cannot be assigned with a different data or class type on the same scope. To detect syntax errors, our compiler performs multiple visits on the ZeroJava code to first extract the classes information, then generate a symbol table, and finally check the static type-safety of all the expressions in the high-level code. Our compiler also throws an error if an `answer` function is missing, as this is required to halt the abstract machine. Consecutively, the ZeroJava compiler parses the high-level code, and generates an IR that is in turn consumed by the code optimizer. In particular, our code optimizer reduces the IR code based on the results of static analysis, employing data-flow analyses and optimization techniques including live-range, dead-code, constant- and copy-propagation [3]. Finally,

our compiler performs register allocation on the IR to further reduce the number of registers and generates zMIPS assembly. Our optimizations based on IR static analysis can be summarized as follows:

- **Live range analysis.** The *liveness* analysis determines which variables hold a value that may be needed in the future (i.e., are live) for each instruction. This is used for the dead code elimination optimization (discussed next).
- **Dead code analysis.** An assignment to a non-live variable is *dead code*; such assignments can be removed, reducing the total size of the program.
- **Constant propagation.** For each program instruction, this analysis determines which variables hold a *constant value*. In this case, the constant value is forwarded to all subsequent uses of the variable.
- **Copy propagation.** Likewise, this analysis determines which program variables are guaranteed to hold identical values. Both constant and copy propagation analyses enable further dead code elimination optimizations.

These optimizations are executed until a fixed point is reached (i.e., a steady state where two consecutive iterations result in the same code sequence); then, no further optimizations can be detected by static analyses. In this work, we employ the Datalog declarative logic programming language from within the IRIS framework [40]. Since Datalog naturally supports recursive relations, it is suitable for fixed-point algorithms [219]. In our case, after the ZeroJava compiler has generated the IR, our code optimizer parses the code and generates relation tables (e.g., simple-instruction, jump-instruction, next-instruction, etc.) that are used for static analysis in Datalog.

Naturally, the object-oriented paradigm comes with a performance trade-off when it is applied to zero-knowledge statements since instantiating new objects requires creating virtual tables and accessing the memory. Therefore, our ZeroJava compiler minimizes any unnecessary memory operations when objects are not used and the statements are only in the main class. Furthermore, the combination of the static

analysis optimizations and register allocation techniques of the ZeroJava compiler are crucial since they minimize the number of registers that are *spilled* (i.e., having to move their values to and from memory). Moreover, as the number of instructions affects the time steps bound, minimizing the total number of zMIPS instructions results in faster proving time.

Debugging. ZeroJava also features a `System.exit(int)` method that can be used to terminate the execution of a ZeroJava program and at the same time return a status code for debugging purposes. Notably, the `System.exit(int)` method is intended to be used solely for debugging and does not replace the `answer` method. Additionally, to enable the advanced debugging techniques of Java, such as breakpoints and the Java debugger (`jdb`), Zilch provides a preprocessor that automatically transforms any ZeroJava program to pure Java code by converting any Zilch-specific statements to the equivalent ones in Java. Specifically, our debugging preprocessor converts the `answer` function to a `System.out.println` invocation followed by a `return` statement, whereas the methods that read from the tapes are replaced with standard Java methods that read from files.

3.2.5 Zilch Back-End Description

Instruction Execution. Each instruction can modify one or more registers, the program counter, and the memory, populating a new row in the transcript, while its corresponding AC defines constraints and assertions for these transitions (both the prover and the verifier agree on these in advance). To ensure correct instruction execution (i.e., *code-consistency*), for each step i the transition between consecutive machine states (S_i, S_{i+1}) is verified by the AC corresponding to instruction i based on the following assertion: Executing instruction i on state S_i results in a new S_{i+1} state, where the destination register in instruction i, as well as the program counter, are updated according to the instruction operation code and all the values on the other registers are propagated to the next state. Each instruction increments the

program counter by one after it is executed, except for `jump` instructions that modify the program counter based on the branch target.

Considering pairs of adjacent states in a time-sorted transcript, code consistency can be checked by inspecting one pair at a time. For instance, after a `move dst, src` instruction is executed, the value in the destination register (`dst`) should be equal to the value of the source register (`src`) before executing the instruction, and all other registers should remain the same. Such constraints should be satisfied between two consecutive states at the execution trace for each `move` instruction. In a `jump` instruction, the consistency of the program counter is asserted while all other registers should remain the same. In a similar manner, we handle the constraints for all instructions that do not involve memory. Initially, the program counter (PC) is set to 0 and the first instruction is fetched; subsequently, each instruction i that is fetched is always pointed by the PC.

Memory Accesses. The back-end of Zilch employs the zk-STARK library to transform the execution trace and the polynomial constraints into a single low-degree polynomial and convince the verifier of their satisfiability over the specific execution trace, which guarantees computational integrity. Similar to code consistency, memory consistency is ensured using constraints on pairs of adjacent states; these states are encoded in a memory transcript sorted by ascending memory locations and then by time. If i is a `load` instruction at a specific address, the value read by i should equal the last value written to that address by the most recent `store` instruction.

By analyzing both the code and memory transcripts, it is possible to verify the consistency of all instructions and memory locations respectively during execution. Specifically, zk-STARK enables $\mathcal{P}$ to convince $\mathcal{V}$ that both transcripts correspond to the same program execution (i.e., they encode the same computation) using a *permutation between the two traces* [29,37]. This permutation is unknown to $\mathcal{V}$ and is verified by zk-STARK via a back-to-back De Bruijn graph, as discussed in [31]. In general, if a program does not use memory-type instructions, the proof comprises fewer constraints,

Table 3.2: zMIPS instructions: R_D denotes the destination register, R_S and R_T denote the source registers, A can be either a source register or an immediate value, while L can be either an instruction number or a label.

Arithmetic Operations		
ADD	R_D, R_S, A	$R_D = R_S + A$
SUB	R_D, R_S, A	$R_D = R_S - A$
MULT	R_D, R_S, A	$R_D = R_S \times A$
DIV	R_D, R_S, A	$R_D = R_S \div A$
MOD	R_D, R_S, A	$R_D = R_S \mod A$
MOVE	R_D, A	$R_D = A$
LA	R_D, L	$R_D = L$
Bitwise Operations		
AND	R_D, R_S, A	$R_D = R_S \,\&\, A$
OR	R_D, R_S, A	$R_D = R_S \mid A$
XOR	R_D, R_S, A	$R_D = R_S \oplus A$
NOT	R_D, R_S, A	$R_D = \sim A$
SLL	R_D, R_S, A	$R_D = R_S << A$
SRL	R_D, R_S, A	$R_D = R_S >> A$
Jumps, Branches and Comparisons		
BEQ	R_S, R_T, L	if $R_S = R_T$ then goto L
BNE	R_S, R_T, L	if $R_S \neq R_T$ then goto L
BLT	R_S, R_T, L	if $R_S < R_T$ then goto L
BLE	R_S, R_T, L	if $R_S \leq R_T$ then goto L
SEQ	R_D, R_S, A	$R_D \leftarrow$True if $R_S = A$
SNE	R_D, R_S, A	$R_D \leftarrow$True if $R_S \neq A$
SLT	R_D, R_S, A	$R_D \leftarrow$True if $R_S < A$
SLE	R_D, R_S, A	$R_D \leftarrow$True if $R_S \leq A$
J	L	goto instruction L
JR	R_S	goto instruction denoted by R_S
Load and Store Operations		
LW	R_D, $A(R_S)$	$R_D = MEM[R_S + A]$
SW	R_S, $A(R_D)$	$MEM[R_D + A] = R_S$
I/O Operations		
PUBREAD	R_D	R_D fetch next word from public tape
SECREAD	R_D	R_D fetch next word from private tape
PUBSEEK	R_D, A	R_D fetch word from public tape[A]
SECSEEK	R_D, A	R_D fetch word from private tape[A]
PRINT	R_S	print R_S
EXIT	R_S	throw exception and return R_S
ANSWER	R_S	return R_S and halt

and its execution overhead can be reduced. Conversely, if the program accesses memory, additional constraints are necessary to verify memory integrity, which can impact

performance; in fact, as the time-bound T increases, the execution time of a program with memory accesses is dominated by the cost of verifying the aforementioned permutation constraints. If all constraints hold during execution and the program finishes within T steps, $\mathcal{V}$ would accept the proof.

zMIPS Assembly Language. In this work, our goal is to define an instruction set architecture (ISA) for the abstract machine of Zilch that is specifically tailored to VC and ZKPK. This means that our candidate instruction set should (a) be sufficiently simple so that the arithmetic circuit corresponding to each instruction would be easy to evaluate, and (b) have a reduced number of instructions so that the number of unique ACs is also minimized. Some modern instruction set architectures, however, such as the x86, implement a large number of instructions that define low-level or compounded operations (e.g., load a value from memory, then multiply it by 2 and finally store it back to memory), or even operate at multiple elements at once. Such complex ISAs are not suitable candidates for our abstract machine; instead, our goal is to define a reduced instruction set computer (RISC) architecture that is compatible with ACs in VC and ZKPK.

For our zMIPS ISA, a natural candidate would be to adopt the MIPS ISA that is sufficiently simple yet very expressive, open-source, and widely used [196]. Moreover, since data memory accesses entail evaluation of additional constraints (as discussed in the previous paragraphs), our ideal ISA should be *register-to-register* and follow the Harvard paradigm with independent memory spaces for instructions and data. Towards that end, we have developed a MIPS-like ISA that includes support for arithmetic, bit-wise, comparison, conditional, memory, and I/O operations. In particular, the zMIPS architecture extends the traditional MIPS ISA with a set of custom I/O instructions for reading public as well as private (witness) data from the input tapes (both sequentially and with random access), as well as instructions to print results and halt.

Instructions. In Table 3.2 we present a subset of the assembly instructions supported by zMIPS. In our notation, register R_D denotes the destination register, while R_S

Table 3.3: ZeroJava Built-in Functions

Built-in function	zMIPS instruction
`Prover.answer(int)`	ANSWER R_S
`System.exit(int)`	EXIT R_S
`System.out.println(int)`	PRINT R_S
`int PublicTape.read()`	PUBREAD R_D
`int PrivateTape.read()`	SECREAD R_D
`int PublicTape.seek(int)`	PUBSEEK R_D, A
`int PrivateTape.seek(int)`	SECSEEK R_D, A

and R_T denote the source registers. Like in the MIPS architecture, our instructions are divided into three broad categories: *R-type* that involves instructions with up to three registers, *I-type* for instructions involving up to two registers and an immediate value, and *J-type* for instructions involving up to two registers and a jump target. In zMIPS, we simplified the MIPS ISA by merging the I and R types, however, we still support the *I-type* instructions (not shown in Table 3.2) for backward compatibility with MIPS programs. Most instructions operate on parameter A, which can be either a source register or an immediate value; in this case, Zilch can distinguish *R-type* from *I-type* automatically. In *J-type* instructions, L denotes either an instruction number or a label. Overall, zMIPS supports arithmetic (i.e., +, -, *, /, `mod`), bitwise (&, |, ˆ, ∼, `<<`, `>>`), logical (!, &&, ||), relational (=, ≠, >, <, ≥, ≤), branch/jump, memory transfer and I/O instructions. Additionally, the 1-to-1 mapping between the ZeroJava built-in methods and zMIPS instructions is summarized in Table 3.3.

Registers. Inspired by the MIPS ISA, zMIPS supports general-purpose ({`$s0`, `$s1`, ...} and temporaries {`$t0`, `$t1`, ...}) and special-purpose registers (SPRs) such as: the `$zero` (or `$0`) register that is hardwired to zero, the `$ra` register that holds return addresses, the stack `$sp` and frame `$fp` pointer registers that are used to enable the call stack of our abstract machine, `$a0` – `$a3` that store call arguments, and `$v0` – `$v1` that store return values. We further introduce the heap pointer `$hp` SPR that is used to store the next free memory address; we utilize `$hp` to perform dynamic memory

allocation in our abstract machine instead of the MIPS system calls.[4] Since zMIPS is an abstract machine, we can increase its total number of registers to more than the 32 used in MIPS. Thus, the abstract machine state comprises a W-bit program counter and up to K registers of size W bits (all initialized to zero); both the word size W and the total number of registers K can be parameterized.

zMIPS Assembler. To enhance the expressiveness of zMIPS, we further introduce the ability to define custom *Macros*, which are new user-defined instructions that are not part of the original ISA. In this case, the zMIPS assembler treats a Macro as a sequence of existing instructions. The latter can improve usability and avoid repetition of instructions since functions and more complex constructions can now be defined as Macros.

Likewise, another assembler enhancement is support for custom *labels* in the code. Specifically, even though the abstract machine assembly instructions use *instruction numbers* as branch targets, the use of labels enables a convenient programming paradigm for users. At the assembler level, our labels are alphanumeric tags that begin and end with a double underscore (e.g., *__a_label__*), while inside Zilch these labels are converted to instruction numbers.

Finally, in our effort to make zMIPS as compatible as possible with the MIPS ISA, we offer support for several assembler expressions, such as the text section (`.text`), and the data section (`.data`). Although these are not used by the Zilch abstract machine, their support renders the zMIPS code backward compatible with MIPS simulators, save for the custom I/O instructions and the absence of system calls.

3.2.6 Application Programming Interface (API) for Zilch

ZeroJava and zMIPS assembly are powerful tools for developing new VC and ZKPK applications; however, additional attention is necessary for existing applications that rely on various system calls and standard library functions. Since our objective is

[4] MIPS invokes `syscall 9` to allocate heap memory. The number of bytes to allocate is passed to the `$a0` register, while `$v0` contains the address of the allocated memory.

to improve the usability of VC and ZKPK in a broad range of scenarios, Zilch further offers a convenient API that allows embedding computational integrity functionality into the code-base of existing C/C++ programs. Using our API, a programmer can independently invoke the prover and verifier of Zilch via C/C++ functions, where each invocation can support arbitrary functionality by passing a zMIPS code snippet to the parent function. In effect, it is not necessary to convert an existing C/C++ application into ZeroJava/zMIPS, except for the specific parts that require computational integrity. The next Section elaborates on our Zilch API, demonstrating two real-life case studies.

3.3 Real Applications in Zilch

3.3.1 Vickrey Auction using Zilch API

To demonstrate the programming interface of Zilch, we implemented a Vickrey auction protocol (also known as *sealed-bid, second-price auction* [233]), in which bidders submit their private bids without knowing the bids of others. As in a traditional auction, the highest bidder wins, but the price paid equals the second-highest bid instead. In the Vickrey protocol, the auctioneer collects a bid and its cryptographic commitment from each bidder, and all commitments must satisfy two basic properties:

- **Binding.** For all non-uniform probabilistic polynomial-time algorithms, the probability of two messages m_1 and m_2 (where $m_1 \neq m_2$) will generate the same commitment c is negligible. Essentially, no bidder can find two different bids with the same commitment.

- **Hiding.** For all non-uniform probabilistic polynomial-time algorithms, the probability of extracting any information about the bid from its commitment is negligible. Hiding ensures that a bidder does not learn anything about the bids of others based on their commitments.

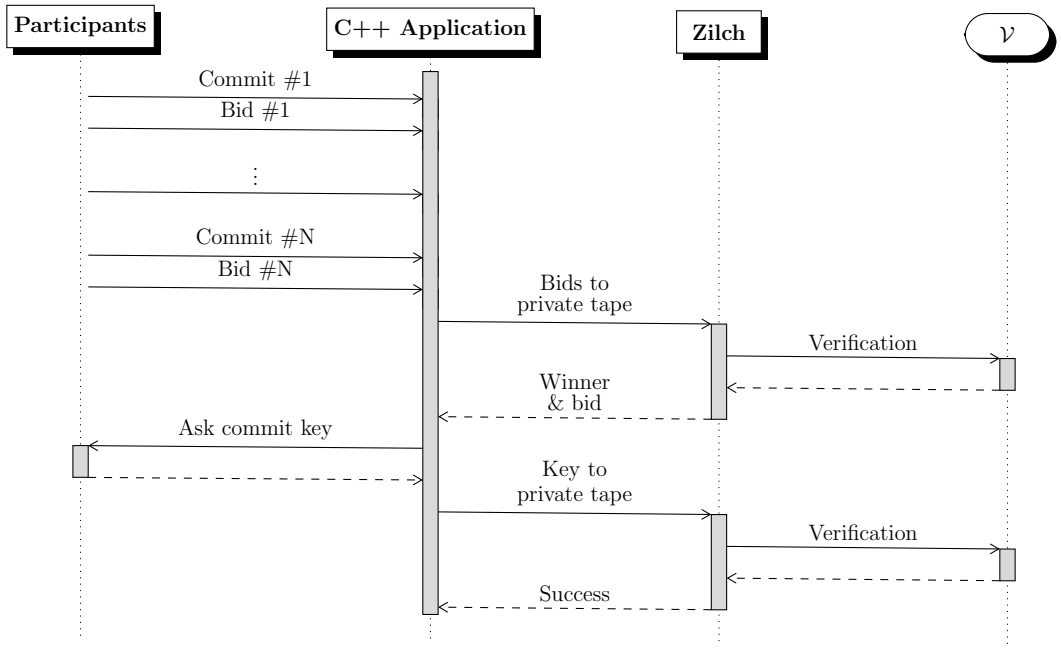


Figure 3.4: Vickrey Auction Overview.

The binding and hiding requirements can be satisfied using a one-way collision-resistant hash function so that recovering a pre-image from the hash output or finding two pre-images with the same output would be intractable.

In our case study, we use the Davies-Meyer (D-M) one-way compression function and implement a single-block Merkle-Damgård hash construction [82] based on a block cipher $\mathcal{E}_k$; specifically, we employ the SPECK cipher with 128 bits block-size and 128 bits key-size [21]. To construct a commitment $\mathcal{C}$, each *bid* value (up to 64 bits, zero-extended) is concatenated with the bidder's commitment *key* (64 bits) and used as SPECK's key input; the bidder's 128-bit random ID (rID) is used as the cipher input to be encrypted and also XORed with the resulting ciphertext, following the D-M construction [82]:

$$\mathcal{C} = rID \oplus \mathcal{E}_{key||bid}(rID) \tag{3.1}$$

For correct execution of the Vickrey scheme, although participants do not have knowledge about the bids of others, at the end of the auction each participant should be able to verify the correctness of the winning bid, even if the auctioneer is not

```
class RangeQuery {
  public static void main(String[] args) {
    int min, max, val;
    val = PrivateTape.read();
    min = PublicTape.read();
    max = PublicTape.read();
    if ((min <= val) && (val <= max)) {
      Prover.answer(true);
    }
    Prover.answer(false);
  }
}
```

Figure 3.5: ZK range query implemented in ZeroJava.

entirely trusted (e.g., the auctioneer may be colluding with a bidder to increase the second-highest bid). Thus, a commitment scheme alone would not be sufficient and computational integrity is necessary to verify the correctness of the protocol.

We implement the auctioneer as a C++ application that collects the individual bids and hash commitments (Eq. 3.1) from all participants, before executing the Vickrey protocol to determine the winner and second-highest bid (Fig. 3.4). The C++ program employs our Zilch API to prove to each participant that the auctioneer function: (a) sorts all bids correctly to find the rID of the highest bidder, and (b) the highest bidder pays the second-highest bid. The latter requires proving computational integrity when the auctioneer opens the committed bids of the highest and second-highest bidders (i.e., verify Eq. 3.1 using $key||bid$ as the witness) and compares these bids with the announced second highest bid; the highest bidder should be convinced that the announced price was actually committed by someone, while the second-highest bidder should be convinced the there is someone that committed a higher bid. During this final step, the highest and second-highest bidder would send their commitment keys to the auctioneer. Overall, the auctioneer's code execution is verified and all bids remain private, except for the second-highest corresponding to the final price.

```
1  secread $t0          # read private input (val)
2  pubread $t1          # read min
3  pubread $t2          # read max
4  move $v0, 0          # result = false
5  bgt $t1, $t0, __end__    # if min > val
6  blt $t2, $t0, __end__    # if val < max
7    move $v0, 1        # result = true
8  __end__:
9  answer $v0           # return result
```

Figure 3.6: ZK range query implemented in zMIPS.

3.3.2 Zero-Knowledge Range Proofs with ZeroJava

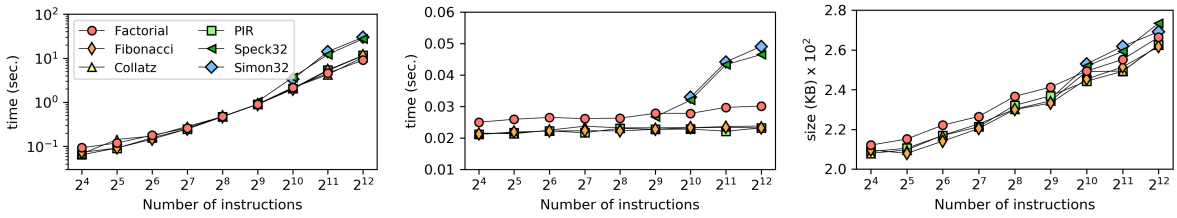
Determining interest rates (e.g., when applying for a mortgage) may require disclosing the credit score of the applicant. Thus, another real-world application with Zilch would be to determine interest rates or loan eligibility while maintaining the privacy of credit scores. Likewise, Zilch can help proving that an account has enough available balance for a transaction, or that an individual is older than 18 years and younger than 65 years without disclosing the exact age. These examples belong to the broader class of zero-knowledge range proofs [153], where Zilch can verify that a secret number is within known bounds without actually disclosing it.

In Fig. 3.5 we illustrate the range query code implemented in ZeroJava, while Fig. 3.6 shows the compiled and optimized zMIPS assembly. Line 1 of the assembly reads the private value *val* (e.g., the age of an individual), while lines 2 and 3 read the lower (*min*) and the upper bound (*max*) from the public tape (e.g., ages 18 and 65 respectively). Consecutively, the program checks that *val* is within the given range (i.e., $min \leq val \leq max$) and returns either 0 or 1.

3.4 Experimental Evaluation

Experimental Setup. We implemented the ZeroJava compiler and optimizer in Java, while the rest of the Zilch framework is implemented in C++.[5] We measured the runtime performance of Zilch using a variety of benchmarks described below. All

[5] The Zilch framework and the ZeroJava compiler are available on GitHub at https://github.com/TrustworthyComputing/Zilch under the MIT license.



(a) Prover time (b) Verifier time (c) Communication complexity size

Figure 3.7: $\mathcal{P}$, $\mathcal{V}$ timings (seconds) and communication complexity size (KB) for a variety of benchmarks for different input sizes and 2^{-60} soundness error. The communication overhead corresponds to the interactive protocol between $\mathcal{P}$ and $\mathcal{V}$.

experiments are obtained on a `t3.2xlarge` AWS EC2 instance running with eight virtual processors up to 2.5 GHz and 32 GB RAM on Ubuntu 20.04.

Multithreaded Prover. The back-end of the Zilch framework is highly parallelizable using OpenMP. It can take advantage of all available threads on the host, and we observe a 2x–4x speedup when using eight virtual cores on AWS.

3.4.1 Our Benchmarks

For our measurements, we adopt the TERMinator suite [185], which comprises scientific benchmarks designed for abstract machines like zMIPS. In particular, the TERMinator benchmarks are beneficial as they do not rely on OS features (such as system calls) while covering a broad range of applications from kernel benchmarks to complex bit manipulations. For our analysis, we implemented the Speck and Simon lightweight block ciphers [21], where the former is oriented towards software implementations and the latter for circuit-based implementations: Speck is based on the Add-Rotate-XOR (ARX) paradigm, while Simon is a balanced Feistel cipher, and both support variable key and block sizes. Being symmetric encryption algorithms, Speck and Simon are very demanding in bitwise operations. Our evaluations also include the Factorial, Fibonacci, and Collatz sequences, as well as the matrix multiplication benchmark, all of which are addition and multiplication intensive. Moreover, a private information retrieval (PIR) program complements our set of benchmarks.

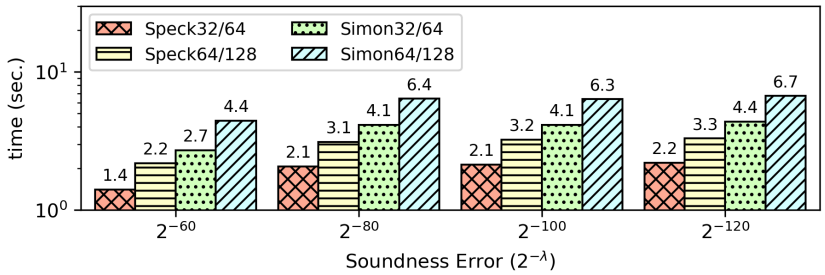


Figure 3.8: $\mathcal{P}$'s measured execution time for the SPECK & SIMON cipher benchmarks using different security parameter sizes on the 32-bit and the 64-bit block sizes with 64-bit and 128-bit keys respectively.

3.4.2 Experimental Results

In our evaluation, we assess the performance of Zilch on a variety of register word sizes (i.e., $W = 8, 16, 32$), as well as different soundness parameters (i.e., $\lambda = 60, 80, 100, 120$). For a soundness parameter λ, the probability that an untrusted prover would violate computational integrity and remain undetected is at most $2^{-\lambda}$. As expected and also confirmed in our benchmarks, larger values for W and λ increase the execution overhead for both $\mathcal{P}$ and $\mathcal{V}$.

In Fig. 3.7 we present the prover and verifier timings as well as the communication complexity sizes for the TERMinator benchmarks and how they scale with an increasing number of instructions (note, while zMIPS is an abstract machine, its instructions are judiciously chosen to map to the MIPS ISA). For each benchmark, we vary the input size accordingly so that the total number of executed instructions matches a power of 2 and show how the prover and verifier timings depend on the number of instructions in the program. Fig. 3.7a shows quasi-linear prover overheads to the number of instructions ($T \cdot polylog(T)$), while SPECK and SIMON incur higher costs because bitwise operations require more complex constraints. Similarly, these two ciphers require poly-logarithmic ($polylog(T)$) verification time to the number of instructions, while the other benchmarks show constant overheads (Fig. 3.7b). Moreover, communication overheads increase linearly to the number of instructions (Fig. 3.7c).

Fig. 3.8 shows the prover's performance on SPECK and SIMON for key sizes 64-128 bits and varying security parameters. As expected, SPECK is faster than SIMON

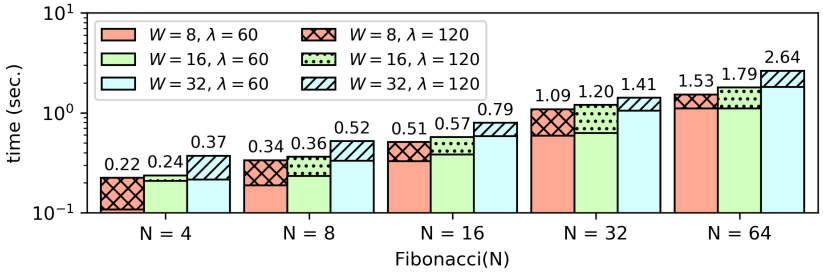
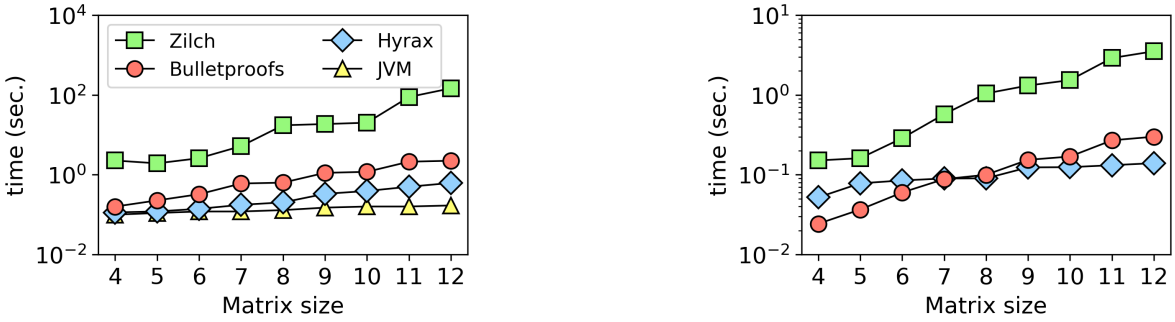


Figure 3.9: $\mathcal{P}$'s measured execution time for the Fibonacci benchmark using different word-sizes $(8, 16, 32)$ and different security parameter sizes for a variety of inputs (2^2 to 2^6).

since the former has fewer instructions and is optimized for software. As λ grows larger, the proving time incurs higher overheads, yet, after 2^{-80} the impact is minimized: using Speck32/64 as an example, an increase of λ from 60 to 80 adds 0.7 seconds to the proving time, whereas increasing λ from 80 to 120 adds only 0.1 seconds. Similar behavior is observed for both ciphers across all configuration sizes.

An overview of the runtime performance of our Fibonacci benchmark for different sizes of W and λ is presented in Fig. 3.9. The bars for $\lambda = 60$ are shown *in front* of those for $\lambda = 120$, and the exact values for the latter are reported. Our experiments show how performance overheads increase with both the input size N (as more instructions are required) and the wordsize W (as more complex constraints are required).



(a) Prover time

(b) Verifier time

Figure 3.10: Comparison between Zilch, Hyrax and Bulletproofs $\mathcal{P}$ and $\mathcal{V}$ timings (seconds) for the matrix multiplication benchmark, as well as the native JVM baseline execution (i.e., without generating a proof).

3.4.3 Comparison with Previous Works

We compare Zilch with 80-bit security with two state-of-the-art transparent zero-knowledge systems: Hyrax [239] and Bulletproofs [61]. Both are based on elliptic curve cryptography and thus their security parameter is not directly comparable with Zilch's. We instantiated them using the M191 elliptic curve [10] over a base field modulo $2^{191} - 19$ giving approximately 90-bit security, based on their reference implementations [235].

For our analysis, we instantiated the standard SHA-256 hash algorithm and compared Zilch with Hyrax for an input block of 512 bits. Our results show that Zilch can prove the correct computation of one SHA-256 block in 73.86 seconds, while Hyrax requires 35.63 seconds; the $\mathcal{V}$ execution time was 1.55 and 1.19 seconds for Zilch and Hyrax respectively. Moreover, in Figs. 3.10a and 3.10b we report the $\mathcal{P}$ and $\mathcal{V}$ timings respectively using the matrix multiplication benchmark and matrix sizes varying from 4x4 to 12x12; as a baseline, Fig 3.10a also shows the native Java matrix multiplication cost (without any proof). Our results show that Hyrax has the fastest performance among the three systems, with Bulletproofs reporting similar timings; in comparison, the $\mathcal{P}$ and $\mathcal{V}$ cost of Zilch is almost one order of magnitude higher than Bulletproofs in matrix multiplication. Likewise, our comparison to the native Java execution shows the performance cost for proving computational integrity across the three systems.

Discussion. The main reason for the observed performance differences between Hyrax, Bulletproofs, and Zilch, is that the computations for the first two have been expressed directly as arithmetic circuits, whereas, in Zilch, the computation was expressed in ZeroJava and then translated to zMIPS using our compiler. While operating directly on arithmetic circuits may achieve faster execution times, using a higher-level abstraction can significantly enhance usability. Notably, the authors' experience working with Hyrax and Bulletproofs showed that it is considerably hard, even for experienced programmers, to develop, debug and analyze any non-trivial program expressed using large monolithic ACs. This limitation informs why our comparisons with these related

works focus on the two pre-compiled circuits already provided by these frameworks. As we discuss in Section 3.5, many related works rely solely on arithmetic circuits. Conversely, Zilch can easily be applied to any computation expressed in our high-level ZeroJava language.

Another important observation is that Zilch inherits from zk-STARK the property of plausible post-quantum security, which cannot be argued for either Hyrax and Bulletproofs. From a security perspective, since Zilch does not require any trusted setup and offers a broader threat model, it is not directly comparable with SNARK-based systems (e.g., [33, 37, 195, 222, 236]) that need a trusted setup; in fact, the total cost of having an offline trusted setup is not directly measurable, as it often includes expensive steps to eliminate the *toxic waste* (e.g., by physically destroying hard drives [248]). In Zilch, our goal is to move to a universal argument system that does not rely on trusted third parties and offers a usable programming model. In Section 3.5, we report further comparisons between Zilch's programming model and those of related works.

3.4.4 Zilch Experiments using our Real-life Case Studies

In this Section, we evaluate the performance and programming complexity for the two real-life applications discussed in Section 3.3 with security parameter $\lambda = 80$ and varying register sizes W.

Vickrey Auction. This application was developed in C++ and linked to Zilch using our C++ API; Speck128/128 was developed in ZeroJava and compiled to zMIPS instructions using the ZeroJava compiler. The word size used for Speck128/128 is $W = 64$ bits, so both the 128-bit key and the 128-bit input block can fit in two registers each. Since this application is interactive across multiple participants, it entails multiple invocations of Zilch using our API (Fig. 3.4): The first invocation iterates on every bid stored in the private tape and performs comparisons to find the winner (highest bid) as well as the amount of the second-highest bid, while additional invocations are required to convince the first and second-highest bidders. In Table 3.4 we show how the

Table 3.4: Vickrey auction: $\mathcal{P}$ and $\mathcal{V}$ times for increasing number of participants with security parameter $\lambda = 80$.

Participants	**Execution Steps**	$\mathcal{P}$ **Time (sec.)**	$\mathcal{V}$ **Time (sec.)**
8	71	0.37	0.025
16	151	1.96	0.026
32	311	4.15	0.026
64	631	8.67	0.027

$\mathcal{P}$ and $\mathcal{V}$ times depend on the total number of auction participants; the former is linear to the number of auction participants, while the latter is almost constant. In this case, since we rely on SPECK128/128 for computing each commitment $\mathcal{C}$ using in the D-M construction (Eq. 3.1), $\mathcal{P}$ performs a new evaluation of SPECK's key scheduling for each different $key||bid$ value of each participant. Each key scheduling requires about the same number of instructions as the SPECK core.

Zero-Knowledge Range Proofs. In our experiments, the high-level ZeroJava code for range-checking (Fig. 3.5) is compiled into zMIPS instructions (Fig. 3.6) using our compiler. This example demonstrates how our programming paradigm in Zilch abstracts all low-level complications and programming complexity for ZKPs, enabling the programmer to express her intent using logical statements very similar to Java. With respect to performance, in this range-checking example, we measured less than 0.1 seconds of prover overhead and negligible verification time, using 16 and 32-bit register sizes.

3.5 Related Work

In the past few years, the interest of the academic community in VC and ZKPs was renewed, leveraging sophisticated cryptography, and interactive and probabilistic checkable proofs. In this section, we discuss several recent works in the area.

Trusted setup per computation. Gennaro et al. introduced in [113] quadratic arithmetic programs (QAP) which inspired many recent works such as Pinocchio [195] and other Succinct Non-Interactive Arguments of Knowledge (SNARKs) [16, 17, 36,

Table 3.5: Comparison of existing ZKP systems based on their cryptographic assumptions, the need for a trusted setup, their universality, and resilience against known attacks from quantum computers. Regarding *ease of programmability*, each bar indicates support for developing ZKPs using arithmetic circuits, assembly language, procedural and object-oriented programming, respectively. Among frameworks that support high-level programming (i.e., those with three or four bars), only Zilch supports the object-oriented paradigm.

ZKP System	Protocol*	Crypto. Assumptions¶	Trans-parent	Univ-ersal	Post-Quantum Resilient	Ease of Program. ACs < ASM < PP < OOP‡	Compiler Available
Pinocchio [195]	zk-SNARK	KoE	○	○	○	▂▄▆▁	○
Geppetto [84]	zk-SNARK	KoE	○	○	○	▂▄▆▁	○
TinyRAM [33]	zk-SNARK	KoE	○	○	○	▂▄▆▁	○
Buffet† [238]	zk-SNARK	KoE	○	○	○	▂▄▆▁	●
ZoKrates† [101]	zk-SNARK	KoE	○	○	○	▂▄▆▁	●
xJsnark† [157]	zk-SNARK	KoE	○	○	○	▂▄▆▁	●
vRAM [251]	zk-SNARG	KoE	○	●	○	▂▄▁▁	N/A
vnTinyRAM [37]	zk-SNARK	KoE	○	●	○	▂▄▆▁	○
MIRAGE [156]	zk-SNARK	GGM	○	●	○	▂▁▁▁	N/A
Sonic [173]	zk-SNARK	AGM	○	●	○	▂▁▁▁	N/A
Marlin [75]	zk-SNARK	KoE, AGM	○	●	○	▂▁▁▁	N/A
PLONK [110]	zk-SNARK	AGM	○	●	○	▂▁▁▁	N/A
SuperSonic [62]	zk-SNARK	ARA	●	●	○	▂▁▁▁	N/A
Bulletproofs [61]	zk-ShNARK§	DL	●	●	○	▂▁▁▁	N/A
Hyrax [239]	zk-SNARK	DL	●	●	○	▂▁▁▁	N/A
Halo [53]	zk-SNARK	DL	●	●	○	▂▁▁▁	N/A
Virgo [249]	zk-VPD	CRHF	●	●	●	▂▁▁▁	N/A
Ligero [7]	zk-SNARK	CRHF	●	●	●	▂▁▁▁	N/A
Aurora [34]	zk-SNARK	CRHF	●	●	●	▂▁▁▁	N/A
zk-STARK [31]	zk-STARK	CRHF	●	●	●	▂▄▁▁	N/A
Zilch† (this work)	zk-STARK	CRHF	●	●	●	▂▄▆▁	●

† These zero-knowledge proof systems focus on front-end optimizations and offer comprehensive programming interfaces.

* SNARK stands for Succinct Non-Interactive ARgument of Knowledge, STARK stands for Scalable Transparent ARguments of Knowledge, SNARG stands for Succinct Non-interactive ARGuments, and VPD stands for Verifiable Polynomial Delegation.

¶ KoE stands for Knowledge of Exponent, AGM stands for Algebraic Group Model, GGM stands for Generic Group Model, ARA stands for Adaptive Root Assumption, DL stands for Discrete Logarithm, and CRHF stands for Collision-Resistant Hash Functions.

‡ ACs stands for Arithmetic Circuits, ASM is Assembly language, PP is Procedural Programming, and OOP is Object-Oriented Programming.

§ Bulletproofs is not considered a zk-SNARK because it is not succinct (i.e., has linear verification time). "Sh" stands for *short* instead of *succinct*.

84, 129, 155, 236, 238, 241]. These protocols, in turn, formed the background for real-world systems as ZeroCash [32]. The proof size is in these constructions is succinct and verification depends on the size of the argument being proven. However, contrary

to Zilch, SNARKs require a trusted and expensive setup phase for every different statement. In many cases, real-world applications that require computational integrity cannot be founded on trusted third parties.

Universal trusted setup. Recent interactive proof-based techniques utilize universal and updatable trusted setups that are based on common – or structured – reference strings. Their advantage compared to the previous category is that they do not require a trusted pre-processing for each circuit, but only a single setup for all circuits. Such constructions include Sonic [173] that composes constant size proofs, as well as PLONK and Marlin [75, 110], which improve upon Sonic by constructing a different polynomial interactive oracle proof (IOP). Although these systems minimize the number of trusted setups to one, the random elements (toxic waste) that are used during this trusted phase may still be used by a malicious prover to forge proofs and break soundness.

Transparent setup. To address the previous limitations, various constructions emerged that are based on different cryptographic assumptions and do not require a trusted setup phase. Bulletproofs [61] and Halo [53] are based on the discrete logarithm problem, while other works such as Ligero [7], zk-STARKs [31], Aurora [34], and Virgo [249] leverage collision-resistant hashes, which can offer additional resilience against known attacks from quantum computers. Likewise, the work in [76] provides a construction that is secure in the quantum random oracle model. Other works such as [123] and [239] are based on interactive proofs. Finally, SuperSonic [62] proposes a new polynomial IOP that relies on groups of unknown orders and does not require a trusted setup. However, a notable limitation of the aforementioned systems is the lack of a practical programming model, which renders the development of ZKPs for arbitrary applications a daunting task.

Random Access Machines. The authors of [33] introduced a random-access machine targeting SNARKs called TinyRAM, which is based on a Harvard architecture. The work in [37] further introduced vnTinyRAM, which is a von Neumann alternative to the original TinyRAM. These TinyRAM variants, as well as vRAM [251], required a

trusted pre-processing phase to generate parameters for verifying different arguments. Conversely, Zilch supports transparent setups where any required randomness is always public and can verify arbitrary programs for any given bound on the number of execution steps, leveraging the state-of-the-art zk-STARK library in its back-end. Notably, our zMIPS ISA offers direct compatibility with existing MIPS programs and enables non-crypto-savvy programmers to easily develop *high-level object-oriented applications* for the zMIPS abstract machine using our ZeroJava compiler and API, whereas this cannot be argued for other random access machines that build their own esoteric models. Lastly, zMIPS supports special-purpose registers, labels, and user-defined Macros, rendering it a comprehensive ISA to ensure computational integrity in general-purpose computation.

Ease of Programmability. In Table 3.5, we present comparisons between various zero-knowledge proof systems. Our comparisons are based on the requirement for a trusted setup, the universality of the ZKP system, the resilience to known attacks from quantum computers, and the ease of developing zero-knowledge proofs from a programmer's perspective.

Pinocchio [195], Geppetto [84], and Buffet [238] provide compilers for translating subsets of the C programming language to arithmetic circuits. In particular, these front-ends cover only a small subset of C (e.g., they do not support pointers) and also require the programmers to deviate from standard C code since they require defining extra constraints, casting statements, and prover-specific types. For example, loops can only have static termination conditions (i.e., cannot depend on non-constant variables) since they are unrolled by the compiler; in effect, this prevents having loops with early termination conditions and programmers must set a fixed bound for each loop. Moreover, while Buffet supports a somewhat larger subset of C compared to Pinocchio and Geppetto, it still lacks support for function pointers, `goto`, and loops with dynamic termination. Likewise, TinyRAM [33] relies on circuit representations that can be generated from C programs; however, the usability of this approach remains limited, as no compiler has been released, and the underlying ZKP protocol does not support

a transparent setup. Similarly, ZoKrates [101] and xJsnark [157] provide front-ends to libSNARK [222] and enable programmers to express a computation using high-level programs that can be translated to arithmetic circuits. Their programming model, however, does not offer support for classes, inheritance, or polymorphism, contrary to the programming paradigm supported by Zilch. More importantly, as summarized in Table 3.5, all aforementioned systems are based on zk-SNARKs and they require a new key generation phase to be invoked by a trusted third party *for each different computation* one wants to prove.

The works of vnTinyRAM [37] and vRAM [251] extend the random access machine introduced in [33] and enable universal circuits that can be used to verify any program up to a given number of machine steps without needing a new setup each time. Similarly, MIRAGE [156] proposes a universal circuit that consumes arithmetic circuits of a bounded number of operations as inputs. Nevertheless, while the generated circuits can implement arbitrary programs, all these works still require an initial trusted phase to set up the circuit. From a developer's perspective, vnTinyRAM employs the same C compiler that TinyRAM does; however, since no implementation of this compiler has been released, developers still have to resort to laborious assembly programming to express their algorithm.

Employing an orthogonal approach, recent zero-knowledge proof systems are also based on universal structured reference strings [75,110,173]. This approach allows a single trusted setup to support all circuits of some bounded size. Contrary to Zilch, these works can only support programming using arithmetic circuits and also require a setup phase by a trusted third party (i.e., they are not transparent).

The bottom section of Table 3.5 includes zero-knowledge proof systems that are transparent (like Zilch). Bulletproofs [61], Hyrax [239], and Halo [53] rely on the discrete logarithm problem, while SuperSonic [62] relies on groups of unknown order and the adaptive root assumption; as a result, these systems remain susceptible to attacks from quantum computers. Regarding their programming model, these proof systems

support computations expressed as arithmetic circuits. Thus, to leverage these systems, a programmer has to manually implement the arithmetic circuit corresponding to their intended algorithm. While the released implementation of Hyrax [235] offers a set of custom scripts to automate this procedure for the arithmetic circuits of the matrix multiplication and SHA-256 examples, the development of new scripts remains as laborious as writing the arithmetic circuits directly. Contrary to Zilch, the programming model of Virgo [249], Ligero [7], Aurora [34] and zk-STARK [31] also relies on arithmetic circuits. While the reference implementation of zk-STARK offers partial support for TinyRAM instructions, however, critical operations such as reading private and public inputs are not supported [38]. Besides, zk-STARK does not offer any compiler to translate high-level programs into STARK proofs. Conversely, Zilch enables programmers to express a computation using our object-oriented ZeroJava language.

3.6 Concluding Remarks

In this paper, we present Zilch, a framework to facilitate the deployment of verifiable computation and zero-knowledge proofs of knowledge for any application. Zilch is transparent (it does not rely on any trusted third-party setup), post-quantum resilient, and using its easy-to-use programming model allows the automated generation of universal circuits that can verify any arbitrary computation for a given time bound. In Zilch, we reduce the problem of proving arguments of knowledge to the granularity of an assembly instruction, so that we can verify instructions independently along with valid transitions between consecutive abstract machine states.

We have designed and implemented the zMIPS abstract machine, a MIPS-like processor model in which each instruction is intricately chosen and translated to a small arithmetic circuit. We complement our framework with a high-level language called ZeroJava and a compiler for translating ZeroJava code into optimized zMIPS assembly instructions. To further improve usability, we have defined a convenient programming API that allows integrating Zilch's prover and verifier into any existing

C/C++ program. In our experiments, we demonstrate the performance of Zilch for a variety of benchmarks, as well as two real-life case studies.

Chapter 4

PRIVACY-PRESERVING IP VERIFICATION

The rapid growth of the globalized integrated circuit (IC) supply chain has drawn the attention of numerous malicious actors that try to exploit it for profit. One of the most prominent targets of such parties is the third-party intellectual property (3PIP) vendors and their circuit designs. With the increasing number of transactions between vendors and system integrators, the threat of IP reuse and piracy has become a significant consideration for the IC industry. What is more, the correctness of 3PIP designs should be verified before integration, imposing another challenge for 3PIP vendors since they have to prove the functionality of their designs to system integrators while protecting the privacy of the circuit implementations. To eliminate this deadlock, we utilize the cryptographic technique of "zero-knowledge proofs" to enable 3PIP vendors to convince system integrators about various functional properties of a circuit (e.g., area, power, frequency) without disclosing its netlist (i.e., *in zero-knowledge*). Our approach comprises a circuit compiler that transforms arbitrary netlists into a zero knowledge-friendly format and a library of modules that provide cryptographic guarantees for various properties of the netlist while hiding the actual gates. We evaluate our method using combinational and sequential circuits from the ISCAS and ITC benchmark suites.

4.1 Introduction

The advent of the Internet of Things (IoT) has made possible an extensive set of applications such as transportation systems, healthcare, home automation, and many more [130]. Because of the demand for small-size and low-power IoT devices, manufacturers have adopted more compact and energy-efficient hardware design paradigms.

As a result, System-on-Chip (SoC) has conquered the market from multi-chip designs by combining lower power and area consumption with increased reliability and functionality, all into a single, integrated chip (IC). In the contemporary IC supply chain, some components are designed in-house and are integrated with a variety of Intellectual Property (IP) cores from third-party vendors in order to fabricate the IC [203].

The ever-growing demand for IC-based solutions results in an increase of third-party IP (3PIP) vendors that try to maximize their profits by providing design standards and guidelines and also by making their IPs reusable so they can be utilized by multiple design layouts [225]. 3PIP cores such as digital signal processors (DSPs) and FFT engines are commonly adopted by chip designers to serve specific purposes in the overall system. This widespread appropriation of 3PIP becomes an attractive target for malicious users and rogue foundries that attempt to trick honest parties and steal their IPs for financial gain [205]. Attackers continue to find novel and elaborate ways to illegally pirate an IP, such as system-level analysis [226] and reverse engineering [69]. Thus, finding solutions to IP piracy and drawing the attention of the IC supply chain to security instead of solely functionality and performance is a major concern.

In the modern IC business model, an essential step of SoC design is IP core verification [72, 95]. First, system integrators (i.e., IP consumers) provide some functional requirements to the 3PIP vendors, who in turn design circuits that meet these specifications. The goal of IP core verification is to prove the functionality of the generated 3PIP designs to system integrators. However, confirming that the circuit complies with the specified properties while achieving high testability is a major challenge in the IC supply chain. Researchers have come up with various solutions, namely application-specific instruction-set processors [221], formal logic verification [141], satisfiability (SAT) solvers [149], and simulation-based methods [177]. Yet, all these solutions focus more on the functional verification of IP designs rather than protecting their privacy. Recent research directions leverage homomorphic encryption to securely outsource the evaluation of the 3PIP netlist to a third party [125, 154, 229]. However, these approaches only protect the input vectors: the netlist designs can still be visible

since homomorphic operations, in this case, do not provide functional privacy (i.e., only data privacy).

Motivated by the lack of netlist-level privacy preservation, we propose a novel method that utilizes zero-knowledge (ZK) proofs to resolve the deadlock of mutual distrust between 3PIP vendors and IP consumers. In this case, the former withholds an IP until a financial agreement is made (due to the IP piracy risk), while the latter refuses to purchase an IP without being convinced that it meets all of the agreed-upon specifications. To instantiate our strategy, we developed the Pythia framework that enables system integrators to verify that a *potentially untrusted* 3PIP vendor possesses an IP that satisfies some agreed-upon properties *without having access to it* (i.e., gaining "zero" knowledge about the IP). One key contribution in Pythia is the conversion of a netlist into a zero knowledge-friendly format that can be evaluated with public test input vectors and generate a public output along with a proof that verifies that the output was computed faithfully.

The back-end of our framework relies on a state-of-the-art ZK protocol called "Scalable Transparent ARgument of Knowledge" (zk-STARK) [31, 184] to implement a library of special state machine modules. These state machine modules can evaluate both combinational and sequential circuits in zero knowledge and further prove various functional properties such as performance, area, and power. Developing these modules as ZK state machines enables Pythia to argue about their computational integrity and offer *provable guarantees* that the IC netlist properties are verified faithfully. Pythia allows the evaluation of 3PIP netlists as Boolean circuits without revealing the netlist itself by encoding input test vectors into a judiciously selected format compatible with our ZK state machine modules. Each state machine consumes a 3PIP netlist as private (i.e., secret) input and test vectors as public inputs; Pythia executes the corresponding ZK state machine on these inputs and generates cryptographic proofs asserting that the output of evaluating the private netlist with the public inputs is correct and computed faithfully.

Overall, in this work, we claim the following contributions:

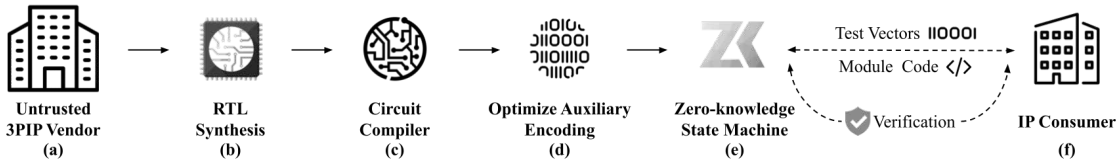


Figure 4.1: Overview of Pythia. (a) The 3PIP vendor ($\mathcal{P}$) possesses an IP described in a Hardware Description Language. (b) $\mathcal{P}$ synthesizes the IP and generates a gate-level netlist. (c) $\mathcal{P}$ determines the evaluation order of the gates and transforms the IP into a zero-knowledge friendly encoding for ZK state machines. (d) $\mathcal{P}$ minimizes the number of intermediate wire values required to evaluate the netlist and divides the encoding into independent shares that can be evaluated in parallel. (e) The 3PIP vendor executes a module (e.g., functional, performance, area verification) with the circuit specification as private input and public test vectors chosen by the IP consumer. (f) The two parties interact and $\mathcal{P}$ convinces $\mathcal{V}$ about the computational integrity of the zero-knowledge evaluation.

- **Circuit compiler.** Development of a compiler that can automatically translate any Boolean circuit (combinational and sequential) into a serialized encoding that can be interpreted by our ZK state machines. Our novel compiler resolves any inter-dependencies between intermediate gate inputs and upstream gate outputs.

- **ZK state machine modules.** Design of a library of ZK state machines that can evaluate Boolean circuits on any input test vector while preserving the privacy of the netlist itself. On top of this functionality, the developed ZK modules can also prove in zero-knowledge various properties of the netlist, such as estimated area, critical path delay, and expected power consumption.

- **Optimizer.** Performance enhancements by utilizing bit-packing and graph-coloring techniques for optimal allocation of the intermediate wires of Pythia's ZK state machine vector. Furthermore, Pythia automatically divides the compiled 3PIP netlist into independent shares that can be evaluated *in parallel*, while cryptographically proving continuity between all these consecutive shares.

4.2 The Pythia Framework

zk-STARKs in Pythia. In this work, we generalize the SHA-256 example (discussed in Section 2.2) and implement a library of custom ZK modules that can evaluate both combinational and sequential circuits as the public computation $\mathbb{A}$. Specifically, each ZK module consumes a secret 3PIP netlist as the witness input $\mathbb{w}$ of $\mathbb{A}$, and a public input test vector $\mathbb{x}$ (provided by $\mathcal{V}$); in effect, algorithm $\mathbb{A}$ simulates the private netlist $\mathbb{w}$ on input $\mathbb{x}$. Each module is tailored to prove different aspects of the target circuit, such as functionality, max frequency, and estimated area/power. Pythia's evaluation output is a public vector $\mathbb{y}$ combined with a cryptographic proof of the correctness of the computation.

4.2.1 Threat Model

In this Section, we elaborate on the different threat scenarios we consider in our approach to address the problem of mutual mistrust between 3PIP vendors and system integrators. Our threat model is twofold: on one hand, we assume a cheating prover $\mathcal{P}$ that does not possess an IP and attempts to trick an honest verifier $\mathcal{V}$, and on the other hand, we assume a cheating verifier $\mathcal{V}$ that tries to learn any information from the communication with $\mathcal{P}$ about the IP.

Cheating Prover. A cheating $\mathcal{P}$ is an adversary with access to the capabilities of a 3PIP vendor and has incentives to deceive the system integrator ($\mathcal{V}$) while attempting to sell an IP that does not meet the agreed-upon functionality. The buyer (system integrator) is expected to test the IP using multiple input vectors and verify the correctness of the outputs, along with the attached ZK proof. The adversary succeeds if she produces a false proof that will convince $\mathcal{V}$ to accept it. Pythia features a configurable security parameter λ that defines the *soundness error* as $\epsilon = 2^{-\lambda}$ and determines the probability that an adversary can successfully deceive an honest $\mathcal{V}$ in the above interactions. Minimizing ϵ is possible by increasing the interaction rounds between $\mathcal{P}$ and $\mathcal{V}$ [30, 31].

Cheating Verifier. Here, a malicious $\mathcal{V}$ is assumed to behave without restrictions and not necessarily follow the protocol specification in order to extract any information about the secret IP. For example, a cheating buyer (system integrator) may attempt to learn the IP netlist before paying, even though the vendor ($\mathcal{P}$) does not wish to disclose the netlist until after payment is received. If $\mathcal{P}$ follows the protocol faithfully, $\mathcal{V}$ will never learn anything about the private witness $\mathbb{w}$ from interacting with $\mathcal{P}$, except the fact that $\mathbb{A}(\mathbb{x}, \mathbb{w})$ is true, which is guaranteed by the underlying ZK protocol. Lastly, we don't consider trivial cases where $\mathcal{V}$ completely withdraws from the protocol between the two parties (e.g., by ignoring messages).

4.2.2 Overview of Pythia

In this work, we present the Pythia framework: a novel method for privacy-preserving 3PIP verification. Pythia utilizes zero-knowledge proofs to enable IP vendors to prove to IP consumers that they possess a 3PIP with some agreed-upon functional specifications without revealing its design. In addition to functional verification, Pythia enables proving in zero-knowledge various circuit properties, such as estimated performance, area, and power. To that end, we have designed a library of ZK state machine modules where each one consumes a netlist as private input and proves a different property of the circuit. An overview of our framework is depicted in Fig. 4.1 and discussed in the following sections.

Privacy-Preserving Functional Verification. The fundamental idea of the Pythia framework is to enable 3PIP vendors (who act as $\mathcal{P}$) to prove to system integrators (i.e., $\mathcal{V}$) that *they possess an IP* with some predetermined specifications. In the first step of our methodology, depicted in Fig. 4.1(a), the 3PIP vendor synthesizes an IP described in a Hardware Description Language (HDL), such as Verilog, and creates a gate-level netlist. Consecutively, $\mathcal{P}$ uses the circuit compiler of Pythia to determine the evaluation order of the gates and resolve any inter-dependencies between gate inputs and outputs from previous gates, before finally converting the IP into a ZK-friendly format that can be evaluated by Pythia's ZK state machine. In the next step, $\mathcal{P}$ utilizes

Pythia's optimizer to minimize the number of intermediate wires required to evaluate the circuit by employing bit-packing and graph coloring techniques. The optimizer also splits the netlist into multiple parallel shares that can be evaluated independently in ZK. Then, $\mathcal{P}$ and $\mathcal{V}$ agree on the algorithm $\mathbb{A}$ that $\mathcal{P}$ will execute (i.e., a state machine that evaluates circuits), and the verifier provides test vectors for the IP that are encoded as public inputs $\mathbb{x}$. Finally, $\mathcal{P}$ executes $\mathbb{A}$ locally to simulate the private netlist (input $\mathbb{w}$) using the public test vector (input $\mathbb{x}$) and computes a public output $\mathbb{y}$ that is sent to $\mathcal{V}$ along with a secure cryptographic proof that the circuit simulation was carried out faithfully.

Library of ZK Modules. Besides functional verification, Pythia supports various algorithms as zero-knowledge state machines, dubbed *ZK modules*, that can be used to prove different properties of the private netlist without revealing any details about it. More specifically, we implemented a library of modules that can assess (a) the expected performance of a circuit by estimating its critical path, (b) the area of a circuit by analyzing the different types and numbers of gates, and (c) the estimated power consumption based on the gate switching activity. Each module is implemented as a different zero-knowledge computation (i.e., as a different $\mathbb{A}$ algorithm) that can be executed within Pythia. Notably, in all aforementioned modules, the input IP netlist always remains hidden from $\mathcal{V}$.

Pythia back-end. The back-end of Pythia utilizes the zk-STARK protocol [31] to argue about the computational integrity of its state machine that simulates a circuit netlist and verifies different properties of it. Internally, Pythia leverages zk-STARK's programming interface to implement the arithmetization procedure and enable expressing a computation as a sequence of state machine transitions along with polynomial constraints that should hold during that computation. Then, Pythia employs these constraints to prove the correctness of the execution and all transitions of the state machine.

In our approach, we implement a circuit simulator as a state machine, and its

state vector is initialized with all zeros. Each step of the circuit simulation algorithm copies the previous state vector, modifies at most one value of it, and appends it after the previous state vector. The generated sequence of state vectors comprises a two-dimensional table that represents the computation (i.e., the *execution trace* in Section 2.2.4). Each new state vector can be updated in a variety of different ways, each of which corresponds to the desired operation that is performed in zero-knowledge. These operations can be arithmetic such as addition and multiplication, bitwise (i.e., conjunction, disjunction, etc.), comparisons, as well as operations that affect the control flow (e.g., a multiplexer). As discussed, our state machine supports two different types of inputs, namely *public* (for the test vectors that $\mathcal{V}$ provides), and *private* that correspond to the witness (i.e., the netlist that only $\mathcal{P}$ knows). Leveraging the security guarantees of zk-STARK [31] in our back-end, Pythia asserts the validity of each transition in the execution trace (each corresponding to a state machine transition) by imposing polynomial constraints and asserts their satisfiability to the verifier $\mathcal{V}$. In effect, Pythia achieves the *seemingly impossible task* of convincing a system integrator about the functionality and various properties of an IP core without ever revealing its composition.

4.2.3 From IP Netlists to ZK-friendly Encoding

Pythia enables automatic compilation of IPs described as HDL programs into a zero-knowledge-friendly format that can be interpreted and utilized by our back-end.

RTL Synthesis. As illustrated in Fig. 4.1b, Pythia synthesizes HDL (e.g., Verilog) programs into netlists consisting of logic gates and primitive memory structures like flip-flops. Without loss of generality, Pythia employs the Yosys Open SYnthesis Suite framework that performs RTL synthesis on an IP implemented in Verilog and generates the corresponding netlist in the Electronic Design Interchange Format (EDIF) [244]. During RTL synthesis, Pythia instructs Yosys to perform various optimizations, like removing unused wires and mapping cells to standard logic gates and small multiplexers.

Combinational Circuits. The generated EDIF netlist is provided as an input to our compiler (Fig. 4.1c), which parses the circuit and identifies all of its gates and wires. Our compiler then performs a second pass in which it associates each gate with specific input and output wires, and uses this information to determine the correct evaluation order of the gates, and eliminate any dependencies between intermediate gate inputs and upstream gate outputs by creating a directed acyclic graph (DAG). Pythia's DAG determines the evaluation order of the circuit's gates by tracking which gates precede each specific gate and then running a topological sort that resolves all of the dependencies in the netlist. After eliminating all dependencies, our compiler serializes the circuit so that Pythia's ZK state machine can prove all gates in sequence. Our compiler also transforms any gates of the circuit that take more than two inputs into a combination of two-input gates; for instance, a three-input `AND` gate is transformed into a pair of two-input `AND` gates. Finally, Pythia assigns a gate identifier to each logic gate (i.e., `AND`, `OR`, `XOR`, etc.) and writes the encoded IP to a file that is used as witness input w by our ZK modules.

Sequential Circuits. Evaluating sequential circuits can be achieved similarly; however, due to complex structures such as the clock signal and flip-flops, they involve a more complicated encoding than combinational circuits. In a software simulation setting where each gate operation amounts to a function call and cannot be "re-used" in the same way that hardware can, sequential evaluation is non-intuitive. The first and most important problem is to deal with the clock. We address this by effectively "unrolling" the circuit for each clock cycle and re-evaluating the gates when the clock ticks. Specifically, *on the first clock cycle* we fully evaluate the circuit and when a flip-flop is encountered, we set its output to 0 and buffer the input. At the start of the next clock cycle, the buffered input to the flip-flop is propagated to the output, and the circuit is re-evaluated. We observe, however, that this method of re-executing the entire circuit on each clock cycle remains inefficient since many gate outputs remain unchanged between consecutive clock cycles. To account for this, Pythia re-evaluates

Algorithm 2 State Machine for Circuit Evaluation

Input: Circuit $\mathcal{C}$ netlist (private), test-vector (public)

procedure EVALUATECIRCUIT
  $\mathcal{H} \leftarrow 0$ ▷ Keeps track of the LCRHF of the IP
  for $idx \in \mathcal{C}_{primary-inputs}$ **do**
    read t from test-vector ▷ Test vector input bit 0/1
    $\vec{S}[idx] \leftarrow t$ ▷ Initialize the state vector ($\vec{S}$)
  for $gate_{ID}, in_1, in_2, out \in \mathcal{C}$ **do**
    $\mathcal{H} \leftarrow LCRHF_{k,k'}(\mathcal{H}, gate_{ID}, in_1, in_2, out)$ ▷ Alg. 1
    if $gate_{ID} = AND$ **then**
      $\vec{S}[out] \leftarrow \vec{S}[in_1] \wedge \vec{S}[in_2]$
    else if $gate_{ID} = XOR$ **then**
      $\vec{S}[out] \leftarrow \vec{S}[in_1] \oplus \vec{S}[in_2]$
    else if $gate_{ID} = \ldots$ **then** ... ▷ NAND, NOR etc.
  return $\vec{S}, \mathcal{H}$

only the gates that depend on upstream flip-flops in the dependency graph. This effectively means that if a flip-flop is encountered on the same path as any given gate in the circuit, this gate is re-evaluated for each clock cycle. This strategy ensures that any gates not connected to and influenced by a sequential circuit element are not needlessly re-computed.

4.2.4 Zero-Knowledge Circuit Evaluation

The fundamental operation of Pythia is the evaluation of Boolean circuits by executing a zero-knowledge state machine on any input test vector while preserving the privacy of the actual netlist. At a high level, Pythia first evaluates all these private gates and also proves the integrity of the computation. The initial state of our ZK machine corresponds to a vector filled with zeros and the state transitions denote the evaluation of logic gates. In each step, Pythia reads a logic gate from the private input, and depending on the type of gate, it determines what operation to perform on the two input wires to compute the correct output. The first level of gates (i.e., the gates with no dependencies) read circuit inputs directly, while intermediate gates depend on the outputs of preceding gates; to evaluate such gates, Pythia further identifies where to read their inputs from. Therefore, after each gate evaluation, we store the output at a *pre-determined index location* of the state vector, which is also encoded in the private

input.

Gate evaluation. For each gate evaluation, Pythia encodes the gate identifier and three state vector indices: two for the gate input values and one for the output of the gate. Moreover, all test vectors and the state machine algorithm $\mathbb{A}$ that evaluates Boolean circuits are public so that both $\mathcal{P}$ and $\mathcal{V}$ can access them; conversely, the gates and the indices used to evaluate each one are only known to $\mathcal{P}$. Although $\mathcal{V}$ is oblivious to the target 3PIP netlist she verifies, $\mathcal{V}$ is convinced that $\mathcal{P}$ has correctly evaluated a circuit using the public input vector, and generated the given public result.

Test vectors. System integrators would typically send many different input vectors to verify that the 3PIP claimed by $\mathcal{P}$ satisfies the predetermined functional specifications. Nevertheless, since our methodology hides the 3PIP from $\mathcal{V}$ by design, the verifier cannot simply trust that $\mathcal{P}$ did not alter the target netlist across evaluations with different test vectors. In other words, a malicious $\mathcal{P}$ could trick an honest $\mathcal{V}$ by using a different netlist each time, in an effort to prove a single 3PIP satisfies the required properties (across all test vectors), when such 3PIP may not actually exist. This is a crucial concern that would violate our threat model (Section 4.2.1). To eliminate any such risks for $\mathcal{V}$, while still keeping the 3PIP private, Pythia employs a secure PRF to generate *authenticated digests* from the circuit during each evaluation. Effectively, this proves to $\mathcal{V}$ that the same exact IP is used across all evaluations with different test vectors. Moreover, since proving the computation of a PRF in zero-knowledge may increase the execution overhead, we employ a lightweight PRF to minimize this impact on circuit evaluation; Pythia employs the PRF described in Alg. 1 with k, k' pre-shared between $\mathcal{P}$ and $\mathcal{V}$.

Circuit evaluation. Alg. 2 outlines the state machine used to prove functional properties of any circuit while evaluating it in zero knowledge. In line 2, we initialize the hash digest $\mathcal{H}$ that Pythia generates during the evaluation of the 3PIP, while in lines 3 and 4 the state vector ($\vec{S}$), which stores the value of every wire in the netlist, is

Algorithm 3 Module to Prove Gate Distribution

Input: Circuit $\mathcal{C}$ netlist (private)

1: **procedure** AREAMODULE
2: $\mathcal{H} \leftarrow 0$ ▷ Keeps track of the LCRHF of the IP
3: $\vec{G} \leftarrow [0, \ldots, 0]$ ▷ Initialize gate vector ($\vec{G}$) with zeros
4: **for** $gate_{ID}, in_1, in_2, out \in \mathcal{C}$ **do**
5: $\mathcal{H} \leftarrow LCRHF_{k,k'}(\mathcal{H}, gate_{ID}, in_1, in_2, out)$ ▷ Alg. 1
6: $\vec{G}[gate_{ID}] \leftarrow \vec{G}[gate_{ID}] + 1$ ▷ Gate type counter
7: **return** $\vec{G}, \mathcal{H}$

initialized with the public test vectors selected by $\mathcal{V}$. Then, the state machine iterates until it evaluates every private gate: First, Pythia reads the gate identifier along with input and output indices (line 6), and updates the hash digest using the PRF from Alg. 1 (line 7). Next, depending on the type of gate, the state machine carries a different operation on the inputs and updates the state vector at each *out* index with the computed output. At the end of the evaluation, Pythia returns the resulting state vector along with the hash digest of the IP as computed by the PRF.

4.3 Library of Modules

Using our state machine for circuit evaluation as a backbone (i.e., Alg. 2), we design a library of auxiliary (AUX) modules that can prove in zero-knowledge various properties of the target 3PIP. In this section, we elaborate on three such modules that focus on area, power, and performance verification, without revealing any details about the corresponding circuit. As discussed in Section 4.2.3, Pythia performs various optimizations during RTL synthesis to remove redundant blocks or unused wires, as well as transform any gates of the 3PIP that have more than two inputs into a combination of two-input gates. Thus, all AUX modules operate on technology-independent reduced netlists.

An important observation is that all our modules process all the gates of the 3PIP regardless of the inputs, resulting in a constant time execution across different test vectors. To prove that the same 3PIP was used as private input for functional verification across all AUX modules, Pythia computes a secure hash digest of the 3PIP

netlist while it parses it. This hash can then be compared against the hash computed by the circuit evaluation module of Pythia, proving that $\mathcal{P}$ did not switch the 3PIP across the different AUX modules.

4.3.1 Area Verification Module

The first AUX module we propose is an area verification state machine. In computer-aided design (CAD) context, area complexity typically refers to the problem of estimating the minimum number of gates required to implement a Boolean function by only having access to the high-level description of the function [159,191]. Therefore, previous works have focused on developing different techniques to estimate the area before implementing a design. Nevertheless, our ZK use case is somewhat different since $\mathcal{P}$ has access to both the HDL description and the EDIF netlist of the IP. In particular, our approach focuses on proving to system integrators that $\mathcal{P}$ owns a 3PIP that meets some functional specifications, and on top of that, it satisfies certain area constraints.

In Pythia, we calculate the area of a circuit by tracking the different gate types and the cardinality of each gate. Our starting point is the circuit evaluation process discussed in Section 4.2.4. Our area verification module, summarized in Alg. 3, utilizes different indices in a gate vector ($\vec{G}$) to track the number of flip-flops (FFs) and each gate type comprising the private 3PIP netlist (e.g., `AND`, `OR`, `NOT`, `XOR`, `NAND`, `NOR`, and `XNOR`). Each time a gate or FF is read from the private input, Pythia increases the corresponding $\vec{G}$ counter by one and then continues to the next. When all gates and FFs have been evaluated, Pythia outputs the hash digest and the $\vec{G}$ value. The former is used to prove that the same private 3PIP was used in this module as in the functional verification case, while the latter contains the values of all counters that can estimate circuit area and track the gate type distribution.

Table 4.1: Logical Effort and Parasitic Delays of Common Logic Gates.

Logic Gate	**Num. of Inputs**	**Logical Effort** g	**Parasitic Delay** p
`NOT`	1	1	1
`NAND`	2	$4/3$	2
`NOR`	2	$5/3$	2
`AND`	2	$7/3$	3
`OR`	2	$8/3$	3
`XOR/XNOR`	2	4	4

4.3.2 Performance Verification Module

The second property we can prove with Pythia using static analysis is the performance of a circuit by calculating the *path logical effort* [220] between inputs and outputs and locating the critical path of a circuit, which is the path with the longest delay. Different gates introduce different delays: for instance, `AND` and `OR` gates have approximately the same delay, whereas `XOR` gates incur higher delays since they may be constructed by a combination of other basic gates [24]. All gate delays are expressed in terms of a basic delay unit τ, which is the delay of an ideal fan-out-of-1 inverter with no parasitic capacitance. We formalize the absolute gate delay as the product of the normalized delay of the gate d and the basic delay unit τ. The normalized delay of an individual gate can be broken down into the summation of the delay of the gate that is dependent on the load f (i.e., *stage effort*), and the delay when the gate is not driving any load p (i.e., *parasitic delay*). Additionally, the stage effort f consists of logical effort g and electrical effort h (i.e., the *fan-out*), such that $f = g \cdot h$. The logical effort is the ratio of the input capacitance of a given gate to that of an inverter capable of delivering the same output current, while the fan-out is the ratio of the input capacitance of the load to that of the gate. Thus, the normalized delay of a single gate can be calculated by the following equation:

$$d = g \cdot h + p. \tag{4.1}$$

We summarize the parasitic delay and stage effort of different logic gates in Table 4.1. Notably, these delays are part of our AUX module configuration and Pythia can easily substitute them with new delays in order to model a specific technology.

In Pythia, we implement the path logical effort methodology as introduced in [220], which is an extension of the aforementioned method for computing the delays of single gates. Given a path, the path logical effort G equals the product of the logical efforts of all the logic gates along the path (i.e., $G = \prod g_i$), whereas the path electrical effort H is the ratio of the capacitance of the last logic gate of the path to the input capacitance of the first gate in the path (i.e., $H = C_{out(path)}/C_{in(path)}$). We also calculate the path branching effort B as the product of the branching efforts at each of the stages along the path. The path effort F is defined as the product of the logical, electrical, and branching efforts of the path (i.e., $F = GBH$) and is used to minimize the delay of a certain path; the latter is minimized when the stage effort in each stage along the path is the same. Finally, the minimum delay achievable along the path is computed by

$$\hat{D} = NF^{1/N} + P, \tag{4.2}$$

where P is the path parasitic delay and equals the sum of the parasitic delays of each gate in the path (i.e., $P = \sum p_i$).

Our Heuristic Method. Minimizing the delay for every single path using the path electrical effort method incurs exponential overheads due to the number of input-to-output path combinations. For large circuits, this approach does not scale gracefully, which motivates the need for tailored optimizations in the context of ZK performance verification. Therefore, we introduce *a novel and efficient heuristic* that uses the single gate logical effort method (i.e., Eq. 4.1) to cascade the delays of the gates of any netlist in a single pass of the circuit and to identify the critical path. In particular, for each gate on a path, our heuristic method propagates the maximum upstream delay of both gate inputs and adds the normalized delay d of that gate. Since a forward parsing[1] of the netlist does not allow computing the exact electrical efforts h of each individual gate, our heuristic assumes that h is an integer equal to the fan-out of the gate (i.e.,

[1] We always analyze the netlist in the same way as with the other modules (i.e., using a forward-pass), to prove the same netlist digest $\mathcal{H}$ is computed.

number of output wires). This assumption significantly reduces the computation cost and makes this method a heuristic; our ISCAS experiments demonstrate the high accuracy of our heuristic, as it can always correctly identify the critical path, while the heuristically computed path delay incurs less than 5% error in all cases.

After we identify the critical path, we apply the exact path logical effort method (i.e., Eq. 4.2) solely to the critical path. Pythia optimizes the aforementioned algorithm to compute the path logical effort of a path by updating all necessary parameters for Eq. 4.2 as we compute the heuristic delay, *requiring only a single pass*. Notably, applying our heuristic method to all paths (Alg. 4, lines 1-3, 6-16, 19-20, 23-25) uses significantly fewer state indices than the exact method of Eq. 4.2 (all lines in Alg. 4), which effectively reduces the number of zero-knowledge operations required by our AUX module.

Exact Method. Alg. 4 summarizes our ZK algorithm $\mathbb{A}$ for heuristically identifying the critical path and calculating the exact path logical effort based on Eq. 4.2 and the formulas from Table 4.1. The exact delay computation is highlighted in blue color and complements the heuristic algorithm (i.e., the black text). Specifically, Pythia processes the gates iteratively (line 9) and stores the heuristic delay in vector $\vec{D}$ (line 20), based on the propagated maximum delay of the two input wires (lines 15 and 16). For exact delay computation, Pythia keeps track of: (a) the path logical effort G by multiplying the logical efforts of single gates along the path (line 21), (b) the path parasitic effort P by adding the parasitics of single gates in the path (line 22), (c) the number of stages in the path N (line 17), and (d) the path branching effort B (line 18). We initialize five vectors to track D, G, B, P, and N with unique counters for every logic gate (lines 3–6). After Pythia has processed all netlist gates, we find the index with the maximum heuristic delay and use it to return the exact path effort $F = GBH$ along with the path parasitic delay and the number of gates in the path.

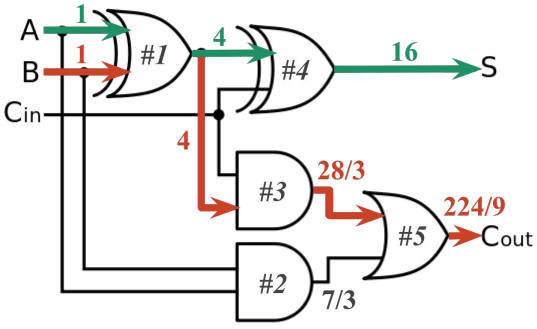
To compute the path branching effort B, we augment the serialized 3PIP netlist with the branching information provided by the Pythia compiler. This creates a unique challenge that we have to address: *"How can we prove that the branching effort that*

Algorithm 4 Module to Compute Path Logical Effort

Input: Circuit $\mathcal{C}$ netlist (private), Path electr. effort H (public)

1: **procedure** PERFORMANCEMODULE
2: $\mathcal{H} \leftarrow 0$ ▷ Keeps track of the LCRHF of the IP.
3: $\vec{D} \leftarrow [0, \ldots, 0]$ ▷ Initialize the heuristic delay vector.
4: $\vec{G}, \vec{B} \leftarrow [1, \ldots, 1]$ ▷ Path logical and branching effort vectors.
5: $\vec{P}, \vec{N} \leftarrow [0, \ldots, 0]$ ▷ Path parasitic delay and gate counter vectors.
6: **for** $idx \in \mathcal{C}_{primary-inputs}$ **do**
7: **read** b from $\mathcal{C}$ ▷ Read branching effort value b.
8: $\vec{h}[idx] \leftarrow b$ ▷ Initialize the fanout vector ($\vec{h}$) for primary wires.
9: **for** $gate_{ID}, in_1, in_2, out, b \in \mathcal{C}$ **do**
10: $\mathcal{H} \leftarrow LCRHF_{k,k'}(\mathcal{H}, gate_{ID}, in_1, in_2, out)$ ▷ Alg. 1
11: $\vec{h}[in_1] \leftarrow \vec{h}[in_1] - 1$ ▷ in_1 is used, decrease b for wire 1.
12: $\vec{h}[in_2] \leftarrow \vec{h}[in_2] - 1$ ▷ in_2 is used, decrease b for wire 2.
13: **assert** $\vec{h}[out] = 0$ ▷ Assert that b at index *out* was correct.
14: $\vec{h}[out] \leftarrow b$ ▷ Update b for current gate at index *out*.
15: **if** $\vec{D}[in_1] > \vec{D}[in_2]$ **then** $max \leftarrow in_1$ ▷ Set max to
16: **else** $max \leftarrow in_2$ ▷ the wire with the maximum heuristic delay.
17: $\vec{N}[out] \leftarrow \vec{N}[max] + 1$ ▷ Count gates in critical path.
18: $\vec{B}[out] \leftarrow \vec{B}[max] * b$ ▷ Update path branching effort.
19: **if** $gate_{ID} = AND$ **then**
20: $\vec{D}[out] \leftarrow \vec{D}[max] + \frac{7b}{3} + 3$ ▷ Update heuristic delay.
21: $\vec{G}[out] \leftarrow \vec{G}[max] * 7/3$ ▷ Update path logical effort.
22: $\vec{P}[out] \leftarrow \vec{P}[max] + 3$ ▷ Update path parasitic delay.
23: **else if** $gate_{ID} = \ldots$ **then** $\ldots$ ▷ cf. Table 4.1 for NAND etc.
24: $idx \leftarrow argmax(\vec{D})$ ▷ Find the index of the maximum value in $\vec{D}$.
25: **return** $\vec{D}[idx], \mathcal{H},$ ▷ Used by both heuristic and exact method.
26: $\vec{G}[idx] * \vec{B}[idx] * H, \vec{N}[idx], \vec{P}[idx]$ ▷ Exact method.

was read from the serialized encoding is correct?". To convince $\mathcal{V}$ about this argument in ZK, Pythia tracks the branching effort of each gate in a separate state vector ($\vec{h}$ in Alg. 4) and decrements the corresponding index every time the output of one gate is used as input by another (lines 11 and 12). In lines 6–8 we read the branching effort of all input wires and initialize $\vec{h}$. When a new gate is processed, we read the branching effort (b) for this gate and check that the previous branching effort stored at that index is indeed zero (line 13). Finally, we update the correct index with the new branching effort (line 14). Alg. 4 proves the correct computation of G, B, P, and N to $\mathcal{V}$, who in turn can compute the path effort F based on the path electrical effort H, which yields the exact path delay $\hat{D}$ using Eq. 4.2. In addition to computing the critical path delay, this module uses the *private netlist input* to construct a hash digest of the 3PIP, so



	G	B	P	N
Initial	1	1	0	0
Gate #1	4; 4	2	4	1
Gate #2	7/3	1	3	1
Gate #3	28/3	2	7	2
Gate #4	16	2	8	2
Gate #5	224/9	2	10	3
Critical	**24.9**	**2**	**10**	**3**

Figure 4.2: Path logical effort calculation. The table depicts how the path logical effort G, path branching effort B, path parasitic delay P, and the number of logic stages in the path N counters are used to estimate the delay of any circuit using the gate delays from Table 4.1. G, B, P, and N are updated based on the type of logic gate and the formulas in lines 17–22 in Alg. 4.

that $\mathcal{V}$ can ensure this hash equals those generated during functional verification and area verification (i.e., the 3PIP was not switched by $\mathcal{P}$).

Example. In Fig. 4.2 we demonstrate how Pythia computes $\vec{G}$, $\vec{B}$, $\vec{P}$, and $\vec{N}$ based on the two inputs and the delays from Table 4.1. On the left-hand side of Fig. 4.2, we illustrate how $\vec{G}$ is calculated: Both input counters of gate *#1* have an initial value of 1 and the output counter gets the value 4 since it is an `XOR` gate. Similarly, the branching effort and parasitic delay of gate *#1* are 2 and 4 respectively. Lastly, N tracks the maximum number of gates along each path. Iterating in a similar manner with the rest of the gates, our method computes the critical path with $G = 24.9$, $B = 2$, $P = 10$, and $N = 3$. The path delays are then computed using the load capacitance H that $\mathcal{V}$ chooses.

Net Delay Estimation. Our aforementioned performance estimation method is based on circuit netlist IPs, so it does not incorporate net delays that depend on placement and routing; as with the standard logical effort method, our module assumes negligible wire capacitance and RC delay. Nevertheless, since interconnection delays are becoming more and more impactful [80], Pythia provides a heuristic to approximate the net delay of the critical path by taking into account the total number of

Table 4.2: Logic Gates Switching Probabilities: P_i denotes that node i is 1 based on input A and B probability being 1.

Gate	**Probability** P_i
NOT	$\overline{P}_A$
NAND	$1 - P_A \cdot P_B$
NOR	$\overline{P}_A \cdot \overline{P}_B$
AND	$P_A \cdot P_B$
OR	$1 - \overline{P}_A \cdot \overline{P}_B$
XOR	$P_A \cdot \overline{P}_B + \overline{P}_A \cdot P_B$
XNOR	$1 - (P_A \cdot \overline{P}_B + \overline{P}_A \cdot P_B)$

wires in the path as well as an optional distribution of wire delays. The latter is agreed between $\mathcal{P}$ and $\mathcal{V}$ and can be derived from past designs of similar size and technology. If no distribution is provided, we assume that all wires in the critical path are "short" and have equal length. In this case, we adjust Alg. 4 output by adding the path net delays based on the average delay per wire, as agreed by $\mathcal{P}$ and $\mathcal{V}$.

4.3.3 Power Verification Module

Pythia also incorporates an AUX module for estimating the dynamic power consumption of a netlist to convince system integrators. Apart from *dynamic* power dissipation, circuits also draw *static* power. The former consists of the switching power (i.e., charging and discharging load capacitances as gate outputs change), while the latter is consumed through various circuit nodes and transistor leakages when a chip is not switching. In this module, we focus on calculating the gate switching activity of 3PIPs, since the static power dissipation is negligible compared to the dynamic power consumption. Various previous works (e.g., [116, 176]) propose different models to estimate the dynamic power consumption of netlists, and the core idea in all of these techniques is to predict the switching activity of the transistors in the circuits. Specifically, the dynamic power of a circuit can be expressed as a function of the probability that a gate transitions from 0 to 1, called *activity factor* α, and the clock frequency f so that $P_{dynamic} = \alpha \cdot C \cdot V_{DD}^2 \cdot f$, where C is the load capacitance and V_{DD} is the positive voltage [243].

Gate switching highly depends on the inputs and thus, the activity factor provides a way to estimate the probability that a gate transitions from 0 to 1 (the reverse transition does not consume any power [243, 5.1.2]), without exhaustively trying different input-vectors, which in many cases is impossible. Since the activity factor depends on the logic function, in Table 4.2 we provide the probabilities P_i that the output of gate i is 1, based on the probabilities of its inputs A and B being 1. We also use $\overline{P}_i = 1 - P_i$ to denote the complement probability (i.e., node i is 0). Using the probabilities of each node, we can derive each activity factor as $\alpha_i = P_i \cdot \overline{P}_i$.

Pythia's power module processes the private 3PIP in a similar manner as in the critical path calculation, yet instead of tracking the different delays, we calculate the activity factor of each gate. All primary input probabilities are extracted from the inputs that $\mathcal{V}$ has provided, and then depending on the gate that Pythia reads, we compute the activity factor of each gate based on the formulas from Table 4.2. For instance, if the input probabilities in Fig. 4.2 are $P_A = P_B = P_{C_{in}} = 0.5$, Pythia estimates the dynamic power consumption of the circuit as follows: The probability that *Gate #1* (i.e., `XOR`) outputs 1 is calculated as $P_1 = P_A \cdot \overline{P}_B + \overline{P}_A \cdot P_B = 0.5$, yielding an activity factor $\alpha_1 = 0.5 \cdot 0.5 = 0.25$. Then, Pythia processes *Gates #2* and *#3* (i.e., `AND`s) and computes $P_2 = P_B \cdot P_A = 0.25$ and $P_3 = P_{C_{in}} \cdot P_1 = 0.25$ which result in $\alpha_2 = \alpha_3 = 0.25 \cdot 0.75 = 0.1875$. Likewise, $P_4 = P_1 \cdot \overline{P}_{C_{in}} + \overline{P}_1 \cdot P_{C_{in}} = 0.5$, and $\alpha_4 = 0.25$, while the probability that *Gate #5* (i.e., `OR`) is 1 is computed as $P_5 = 1 - (\overline{P}_3 \cdot \overline{P}_2) = 0.4375$, producing an activity factor $\alpha_5 = 0.4375 \cdot 0.5625 = 0.246$. Instead of arbitrarily setting the input probabilities, we employ a probabilistic model that extracts the primary input probability distribution based on $\mathcal{V}$'s test vectors. Essentially, the probability of each input is calculated as the average of a specific input across multiple test vectors. As in all library modules, Pythia computes a secure hash of the private netlist in ZK, so the Verifier can compare it with the one calculated during functional verification to be convinced that the estimated power consumption was calculated over the same 3PIP.

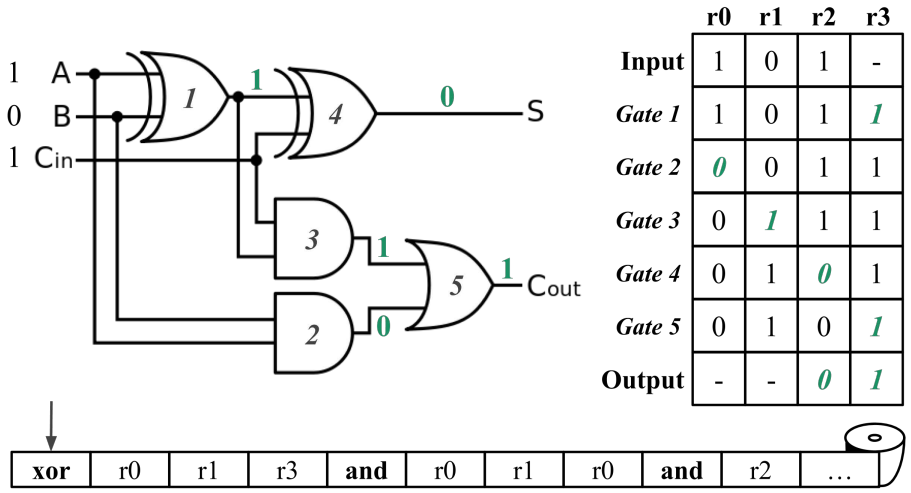


Figure 4.3: Adder evaluation. The numbers on the gates denote the evaluation order, also illustrated by the row labels of the table (execution trace) on the right-hand side. The green values represent which variable changed after the evaluation of the gate denoted by the row number. The variables $r0$–$r3$ can be interpreted either as four separate indices or as one 4-bit block. The serialized private input encoding a part of the above circuit is depicted at the bottom.

4.4 Pythia's Optimizer

In this Section, we propose a series of optimizations to effectively reduce the size of the state vector, since a smaller state vector results in improved performance for both $\mathcal{P}$ and $\mathcal{V}$. Finally, we introduce a methodology to split the functional and property verification modules to exploit parallelization.

4.4.1 Efficient Wire Placement Using Register Allocation

Apart from the area verification module that uses a constant number of state indices to keep track of the numbers and types of gates for each different netlist, the other modules require more indices as the total number of gates in the netlist increases. For small circuits, assigning each wire to a unique state vector index is feasible, however, for bigger circuits the size of the state vector required to hold all of these wires grows significantly, impacting both the time required to generate a proof and the time to verify it. Therefore, we apply a tailored *register allocation technique* that utilizes graph coloring to minimize the number of unique indices needed to hold

the intermediate wires of a circuit. We incorporate this optimization in the Pythia compiler during the IP transformation phase: instead of assigning unique indices to each wire, Pythia deftly allocates and re-uses indices for the input, intermediate, and output wires in the state vector.

Fig. 4.3 demonstrates how the register allocation technique effectively reduces the number of unique indices and how they can be reused to prove the functionality of an adder. Without this optimization, the circuit in Fig. 4.3 would require 3 input (i.e., A, B, C_{in}), 2 output (i.e., S, C_{out}), and 3 intermediate wires (i.e., to hold the results of gates 1, 2 and 3), resulting in a total of 8 different indices. In Fig. 4.3 we demonstrate how our graph coloring technique allocates each wire to an index and enables Pythia to evaluate the adder with solely 4 indices. The table on the right-hand side of the figure depicts the execution trace of the adder (i.e., how the state changes with each gate evaluation). In the first row of the execution trace the state vector is initialized with the values of the primary inputs: $A = 1$, $B = 0$, and $C_{in} = 1$. In the second row, gate #1 is evaluated and its output is written at the last index of the state (highlighted in green). Each row corresponds to the evaluation of the next gate, while the last row holds the two outputs of the adder. At the bottom of Fig. 4.3, we show a part of the 3PIP witness input w that the Pythia compiler generated from the adder circuit using register allocation. First, the state machine reads the `XOR` gate and the two input variables (`r0`, `r1`), then it evaluates the gate using the contents of the two indices (i.e., 0 and 1) and stores the result to the output variable (`r3` = 1). Then, our state machine continues to the next operation. This optimization significantly reduces the total number of required indices, and as we show later in Section 4.5, this is crucial for the performance of Pythia.

4.4.2 Bit-Packing

Large combinational and sequential circuits that consist of thousands of wires and gates require a considerable amount of indices to hold intermediate wire values even when register allocation has been applied. Register allocation techniques are not

as effective in circuits with many gates at the same level (i.e., circuits with large width) since all these wires should hold their values at the same time. An important observation about our functional verification module is that each state index stores only a binary value, which yields state vectors with many indices. Notably, increasing the number of indices in the state vector impacts the number of zk-STARK operations, which can affect performance. Therefore, *to minimize the number of indices*, yet continue storing the same amount of information, we employ a *bit-packing* optimization that organizes multiple wires into multi-bit blocks under the same state vector index. For instance, using this bit-packing optimization, the state in Fig. 4.3 becomes a single 4-bit register R_0 instead of four 1-bit registers (r_0 to r_3). In this example, R_0 will transition as follows: $1011 \rightarrow 0011 \rightarrow \cdots \rightarrow 0101$.

Specifically, the bit-packing scheme of Pythia can utilize uniquely indexed 64-bit blocks, which reduces the number required by a factor of 64. Notably, Pythia can still access individual bits within each 64-bit block. To enable this optimization, Pythia's compiler can encode both the block index within the state vector and the bit index within the block for the inputs and output of each gate. Therefore, in addition to the previous four inputs (i.e., $gate_{ID}, in_0, in_1, out$ in line 6 of Alg. 2), each gate evaluation includes three-bit indices as well (i.e., one for each input and one for the output). Although this feature seemingly complicates Pythia's state machine as it incurs bit-shift operations to read/write individual bits within a block, this overhead is negligible compared to the ZKP cost savings from having fewer state vector entries. This optimization is mostly applicable to functional verification, as the power and performance modules already store multiple bits in each index, while the area module uses a constant state vector size so bit-packing does not impact its performance.

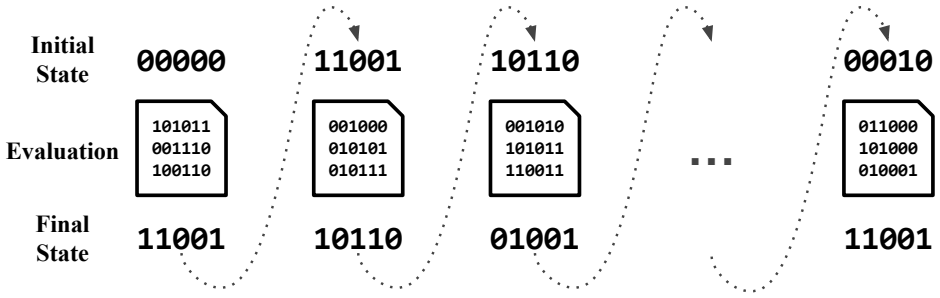


Figure 4.4: Chaining execution to enable parallel verification. Pythia divides large executions into multiple shares and pre-computes the intermediate states locally. The simulator computes the PRF digest of the machine state and compares it with the digest provided in the public input to verify its integrity. Each share can be verified independently and in parallel.

4.4.3 Execution Parallelism

To further optimize the performance of our methodology, we augment Pythia's optimizer to enable parallelized proof generation and verification. Specifically, we observe that all four Pythia modules operate iteratively over a given netlist. The execution trace of each module is initialized with a zero state vector and transitions to different states during the computation; given any intermediate state and a private 3PIP, Pythia can always continue the computation and converge to the correct output. Therefore, we can break down the problem of proving one big execution trace into the problem of proving the faithful execution of two smaller execution traces, along with a provable transition between the two. We refer to each smaller execution trace that is a part of a bigger trace as a *share*. We demonstrate this concept in Fig. 4.4, where the sample initial state of the first share is 00000, and after the evaluation of the share its state becomes 11001. The latter initializes the second share, which will produce a new state 10110, and so on. The final state 11001 represents the result of the computation, which can be either the output wires of a netlist (functional verification), or different counters encoding the area, performance, or power estimations.

Although the shares in Fig. 4.4 are constructed honestly by the 3PIP vendor, $\mathcal{V}$ is not able to immediately verify if the intermediate states are initialized correctly or a malicious $\mathcal{P}$ has modified them. To convince $\mathcal{V}$ that these shares are actually parts

of the original execution trace, and also preserve the confidentiality of the intermediate state vectors, we compute integrity measurements of each state vector using the lightweight PRF discussed in Section 2.3. The 3PIP vendor calculates these publicly authenticated digests and shares them with $\mathcal{V}$; notably, only $\mathcal{P}$ knows the actual netlist data that produced each digest. At last, the system integrator verifies that all PRF digests at the end of each share are the same as the ones used at the beginning of each next share.

So far, we demonstrated how to decompose a big execution trace into smaller consecutive shares, and chain them in order to compute the correct output. Nevertheless, to exploit parallelism in zero-knowledge we also need to pre-compute the intermediate state vector values. In this case, Pythia quickly evaluates the netlist without creating a ZK proof, and computes the starting and ending state vectors of each share, along with their PRF digests. The offline execution overhead for $\mathcal{P}$ is negligible compared to the online proof generation timing. This method is not affected by any dependencies between consecutive shares, as all intermediate state vectors and their hash digests are precomputed, so proof generation of each share can take place in parallel. Finally, $\mathcal{V}$ can verify that the public PRF digests at the end of each share match the ones used to initialize the next share to ensure correctness.

4.5 Experimental Results

4.5.1 Experimental Setup

In this Section, we evaluate the applicability and performance of our framework using selected benchmarks from the ISCAS'85, ISCAS'89, and ITC'99 suites. To verify the correctness of our circuit evaluation (Alg. 2), we employ a variety of input-output pairs for our benchmarks, as well as two arithmetic cores from [230]: an 8-bit high-radix sequential multiplier and a 12-bit sequential fast modular reduction core for Mersenne prime moduli. We synthesize the benchmarks to produce EDIF netlists using the Yosys Open SYnthesis Suite framework [244]. We used Verilog and VHDL implementations for the ISCAS and ITC benchmarks, respectively. Yosys features various parameters

that produce different netlists depending on the user's preferences (e.g., optimize for performance, space, etc.). In this work, we synthesized our benchmarks with the `proc` and `flatten` flags. Notably, more sophisticated tools may result in more optimized netlists, however, our goal in Pythia is to optimize the cost per gate for $\mathcal{P}$ and $\mathcal{V}$, rather than to optimize the benchmarks themselves.

The core of the Pythia framework is implemented in C++, while Pythia's compiler and optimizer are implemented in Python 3. To fully investigate parallelization, we obtained our experimental results on an m5.24xlarge AWS EC2 instance with two Intel Xeon Platinum 8175M cores running at 2.5 GHz and hyper-threading, resulting in 96 virtual cores and 748 GB RAM. The system is running Ubuntu 18.04 with the 4.15.0 Linux kernel and the g++ 7.4.0 compiler.

4.5.2 Performance Evaluation.

The timing and memory overheads incurred by Pythia's back-end highly depend on both the length of the execution trace and the size of the state vector. As the execution trace becomes bigger, both the execution timing and the memory required for $\mathcal{P}$ increase. The verification timing is also affected by the same factors, however, it is always poly-logarithmic ($polylog(T)$) to the size of the execution trace T. An important property of Pythia's back-end (inherited from zk-STARK) is that the proving cost for any two execution traces whose size is less than the same power of 2 is approximately the same [31]. For instance, a trace with 942 steps incurs roughly the same timing and memory overheads as a trace that has 1020 steps since both have less than 2^{10} transitions.

Splitting Shares Trade-off. In this experiment, we study the optimal size for our shares by investigating the number of gates in each share. Increasing the number of shares allows proving more shares in parallel, however, the cost of calculating the PRF digest in every share will eventually dominate the proving time. Conversely, having fewer shares makes the cost for the PRF negligible compared to the total cost of proving the evaluation of each share, while the required memory increases as well. In Fig. 4.5

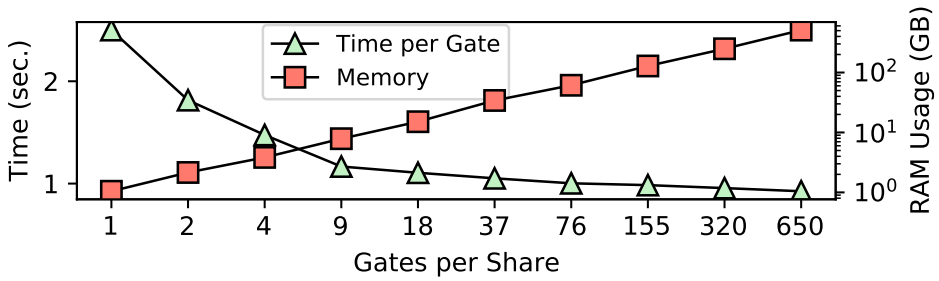


Figure 4.5: Memory/Efficiency per gate trade-off for the prover. The horizontal axis shows the maximum number of gates per share that can be evaluated in less than a power of 2 state machine transitions.

we illustrate the aforementioned trade-off based on the proving time per gate for our functional verification module: the *left* vertical axis shows the time per gate, while the *right* vertical axis indicates the required memory for each splitting configuration. While the time for $\mathcal{P}$ increases as the execution trace becomes longer, having more gates in each share reduces the proving time per gate. The green triangle trend in Fig. 4.5 indicates that increasing the number of gates per share (i.e., having fewer shares) decreases the overall proving time; however, the associated memory cost (red squares in Fig. 4.5) scales linearly with the number of gates per share. Therefore, the cost for $\mathcal{P}$ depends on the number of shares that can fit in the available memory of the target host. For our experimental setup, we identified that 9 gates per share strike a good trade-off between proving time and required memory.

Two levels of Parallelization. Pythia is *doubly* parallelizable: (a) using multiple independent shares, we can split the computation and evaluate the shares in parallel, and (b) Pythia's back-end utilizes multiple threads to parallelize the proof generation in each share. This motivates our next experiment which investigates how we can optimize parallelism in a multi-core host. The green triangles in Fig. 4.6 present the proving time for 1 share using a different number of cores. We observe that having more than 8 cores per share reduces the relative speedup (diminishing returns), so the cores per share should be balanced with the number of shares processed in parallel on all available cores.

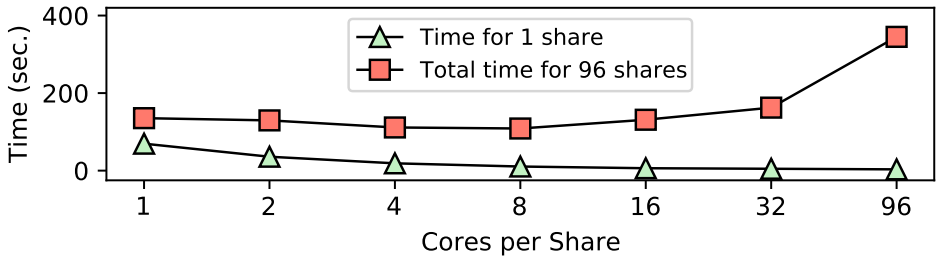


Figure 4.6: Time measurements for proving 1 and 96 shares with a different number of threads per share. The red squares depict the timings for 1 core/share (prove all 96 in parallel) to 96 cores/share (prove shares sequentially).

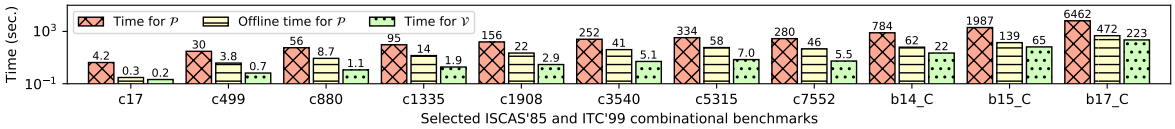


Figure 4.7: $\mathcal{P}$ and $\mathcal{V}$ experimental results for selected benchmarks from the ISCAS'85 and ITC'99 suites.

Moreover, the red squares in Fig. 4.6 investigate the total cost for proving 96 shares by varying the number of allocated cores per share. On one extreme, we can assign 1 core per share in order to verify all 96 shares in parallel (i.e., leftmost red square in Fig. 4.6), while on the other extreme we can assign all 96 cores to verify 1 share at a time and repeat the process 96 times (rightmost red square). If we allocate 8 cores per share, we can verify 12 shares in parallel per iteration (note, we need 8 iterations to verify all 96 shares), our setup achieves the fastest overall time for $\mathcal{P}$, which further confirms the finding of the previous paragraph.

Discussion of Experimental Results. We evaluate Pythia using the ISCAS'85, ISCAS'89, and ITC'99 benchmark suites [59, 60, 81] with random input test vectors (chosen by the IP consumer). Specifically, $\mathcal{V}$ is responsible for choosing multiple test vectors to supply to Pythia and this selection does not impact the proving time. We observe that each benchmark's proving time depends on the total number of gates in the netlist, rather than the distribution of the input pattern, even though different inputs trigger different gates. In Figs. 4.7 and 4.8, we report our performance evaluation

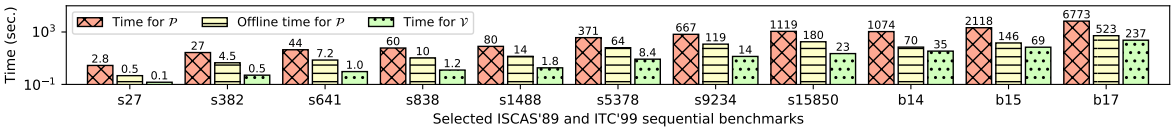


Figure 4.8: Amortized $\mathcal{P}$ and $\mathcal{V}$ evaluation time per cycle (over 10 cycles) using selected benchmarks from the ISCAS'89 and ITC'99 suites.

results for a selection of benchmarks, using a splitting of 9 gates per share and 8 cores allocated per share. In particular, Fig. 4.8 shows the amortized cost per cycle (over ten clock cycles) for each of our sequential benchmarks. As discussed in Section 4.4.3, in order to prove multiple shares in parallel, $\mathcal{P}$ needs to quickly compute the intermediate state vectors offline (i.e., without generating a proof). Therefore, each benchmark in Figs. 4.7 and 4.8 reports the aforementioned offline cost (using the yellow bars in the middle), as well as the online costs for both $\mathcal{P}$ and $\mathcal{V}$ using the red (left) and green (right) bars, respectively. All timings depend on the length of the execution trace, which in turn depends on the number of gates in each netlist share; in this case, all shares comprise exactly 9 gates, so the trace length is constant. Thus, the factor that dominates Pythia's performance is the number of shares in each benchmark (i.e., the netlist size).

In the Verilog implementation of the ISCAS'85 benchmarks, `c7552` comprises more gates than `c5315`; however, after Pythia invokes Yosys to synthesize and transform all the gates with more than two inputs into a series of two-input gates, `c5315` has the most gates compared to the other ISCAS'85 benchmarks, and thus incurs the higher $\mathcal{P}$ and $\mathcal{V}$ costs. We report the number of gates and wires of our synthesized ISCAS benchmarks in Table 4.3. One can observe that the proving time in Pythia scales linearly with the number of shares (i.e., $\lceil \#gates/9 \rceil$) which is quasi-linear ($T \cdot polylog(T)$) in the number of state machine transitions T since each gate involves a fixed number of transitions. Likewise, Pythia's verification time is poly-logarithmic ($polylog(T)$) in T.

Fig. 4.9 reports our experimental evaluations for both $\mathcal{P}$ and $\mathcal{V}$ using our area,

Table 4.3: Number of Gates and Wires Generated by Pythia Compiler and State Vector Minimization After Applying Graph-Coloring and Bit-Packing Techniques for Selected Benchmarks.

Combinational Benchmark	**c499**	**c880**	**c1355**	**c1908**	**c3540**	**c5315**	**c6288**	**c7552**	**b14_C**	**b15_C**	**b17_C**
# Gates	246	583	1006	1435	2349	3454	2922	2742	4447	7901	24246
# Wires	287	643	1047	1468	2399	3632	2954	2949	4722	8384	25696
Vector Size	48	87	72	189	322	569	63	357	574	620	1791
64-bit Blocks	1	2	2	3	6	9	1	6	9	10	28

Sequential Benchmark	**s349**	**s382**	**s420**	**s444**	**s526**	**s641**	**s838**	**s1488**	**b14**	**b15**	**b17**
# Gates	234	292	322	346	382	455	658	843	4338	8322	25697
# Wires	247	299	314	353	389	494	696	855	4369	8357	25733
Vector Size	57	78	83	82	132	78	170	242	772	1015	3756
64-bit Blocks	1	2	2	2	3	2	3	4	13	16	59

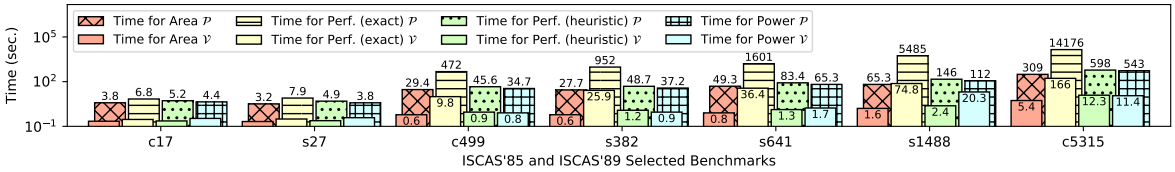


Figure 4.9: Timing results for $\mathcal{P}$ and $\mathcal{V}$ of the area, performance (for both exact and heuristic methods), and power modules for selected ISCAS'85 and ISCAS'89 benchmarks. The timings for ISCAS'89 netlists are amortized over 10 cycles. $\mathcal{P}$'s offline costs are reported in Figs. 4.7 and 4.8 and are omitted from this plot. Our heuristic method incurs less than 5% error in all cases, and offers significant performance benefits compared to the exact method.

performance, and power estimation modules. As mentioned in Section 4.4.2, the bit-packing optimization cannot be applied to the performance and power estimation modules since they already store multiple bits of information in each state index. As a result, these two modules incur higher overheads than the functional verification module due to the significantly larger state vectors they use. Our performance module using the *exact method* for path delays requires significantly more state indices than any other module to keep track of all wire delays and therefore incurs considerably higher overheads. Nevertheless, *our heuristic method* achieves highly accurate results (less than

5% error from the exact path delays of the ISCAS benchmarks), yet it is several orders of magnitude faster. Likewise, across several ISCAS'85 benchmarks, our heuristic incurs less than 4% error (on average) compared to the results from OpenTimer, a popular open-source static timing analysis tool [133]. Finally, since the area verification module has minimal requirements (i.e., small and constant number of indices for all the netlists), it achieves the fastest performance across all the modules. Moreover, as we observe in the results of Fig. 4.9, the overhead of our AUX modules scales with the total number of gates in each netlist, while the prover's performance also depends on the number of indices of the state vector.

Bit-Packing. Table 4.3 summarizes the numbers of gates and wires for our ISCAS benchmarks after Yosys synthesis and demonstrates how our register allocation and bit-packing optimizations drastically reduce the number of intermediate wires. Notably, register allocation is less impactful for very wide netlists, as Pythia maintains all wire values at each circuit level using unique state indices, which attributes to increased overheads. For narrower netlists that have smaller levels (regardless of their depth), our register allocation can significantly reduce state indices and hence Pythia's overheads. For instance, the netlist of `c6288` has 2954 wires, and register allocation can reduce them to just 63, which is a decrease by almost 50× (c.f., row "Vector Size" in Table 4.3). Conversely, netlist `s526`, which achieves a modest decrease of about 3× according to Table 4.3, would benefit less from this optimization. When our bit-packing technique minimizes the state vector size (c.f., row "64-bit Blocks" in Table 4.3) to a value less than 10, the performance benefits are more noticeable.

4.6 Related Work

In this Section, we discuss the different categories of zero-knowledge proof systems and our choice of zk-STARK as the back-end cryptographic protocol of Pythia. The first category involves systems that need a *trusted setup phase for each different computation* they want to prove. This line of work started from the construction proposed by Gennaro et al. [113] using quadratic arithmetic programs and continued with

the works in [33, 37, 195]. More recent proof systems utilize *universal and updatable trusted setups* that are based on common reference strings (e.g., [75]). Their advantage compared to the previous category is that they do not require a trusted pre-processing phase for each circuit, but only a single setup for all circuits. Although this method requires only one trusted setup, neither this nor the first category are applicable for our threat model since a malicious party that gains access to the trusted setup phase can always forge false proofs.

The last category includes systems with a *transparent setup* (i.e., no requirement for a trusted third-party to initialize the system). This includes preliminary works in [51, 61, 239], as well as zk-STARK-based systems [31, 184]. Pythia's back-end relies on zk-STARK as it supports universal, trustless setup in line with our threat model; moreover, verifiers in zk-STARK protocols can achieve the best-in-class performance, while the low-overhead proving times [31].

The authors of [154] proposed a novel technique that leverages homomorphic encryption (a cryptographic primitive that allows performing meaningful operations on encrypted data) to enable secure outsourcing and evaluation of 3PIP designs using encrypted input vectors. In a similar direction, the authors of [125] propose a framework that converts Verilog HDL programs into homomorphic circuits with equivalent functionality, and evaluates them using encrypted input vectors. Both of these approaches allow untrusted third parties to evaluate an IP while preserving the privacy of the input test vectors. Nevertheless, since homomorphic operations only protect data privacy but not the applied function (i.e., Boolean gates remain unencrypted in the source file), the netlist designs are not protected as in Pythia.

Previous work has also focused on proving the correctness and various properties of 3PIP designs using formal logic verification [141, 150]. These techniques mostly focus on formally checking the correctness of a netlist or a cryptographic protocol to ensure the design is free of malicious logic, such as hardware Trojans (e.g., [227]). Yet, such methods focus on comparing the netlist with a formal mathematical model of the circuit and exclude any privacy protections for the IP. Conversely, the threat model

of this work (Section 4.2.1) assumes that cheating verifiers have incentives to extract information about the IPs before a financial transaction is completed. In this context, Pythia offers a novel privacy-preserving approach to such deadlocks, by enabling system integrators to verify that 3PIP vendors possess an IP with certain functional, performance, area, and power constraints.

Recent research directions have also focused on obfuscation methods that aim to prevent IP theft of circuit designs, by inserting additional gates into the circuit in order to hide its implementation [205]. Likewise, watermarking [144] and fingerprinting [64] methods embed author signatures in the design to deter IP theft, as they allow tracking the source of leakage in case of piracy violations. While such techniques mitigate the risk of IP theft, they come with important limitations: First, these techniques only make it harder for the attackers but do not guarantee that 3PIPs can not be leaked. Furthermore, all the aforementioned techniques tamper with the IP design and, more importantly, they assume that verification takes place *after the IP is outsourced* to the system integrator. In contrast, Pythia is designed for privacy and makes it impossible for such designs to be leaked in the first place, as system integrators never access the IP while verifying its properties. Notably, in this work, we can prove both the functionality of a 3PIP, as well as estimate important properties (area, performance, and power) *without altering the netlists* in any way. To the best of our knowledge, Pythia is the first framework that employs ZKPs in the context of privacy-preserving IP verification, which is an exciting research direction for integrated circuits.

4.7 Concluding Remarks

In this paper, we have proposed the first-of-its-kind Pythia framework for privacy-preserving IP verification. Pythia features a custom compiler that translates any circuit into a specialized encoding that is then evaluated in zero-knowledge using public test vectors; our method generates cryptographic proofs to attest the faithful evaluation of test inputs on a netlist, yet the netlist itself remains a black box for the verifier. Security is guaranteed by Pythia's back-end that leverages the provably secure zk-STARK

protocol. Our methodology also supports different auxiliary modules that can further estimate the area, performance, and power consumption of a netlist in zero knowledge. Moreover, Pythia leverages authenticated hash digests to prove that the secret netlist was not altered across different test vectors. Our methodology is complemented by optimization techniques that reduce performance overheads and memory requirements using register allocation and bit-packing. Notably, Pythia can automatically split a given netlist into multiple shares that can be evaluated in parallel. Finally, we have demonstrated Pythia's versatility using combinational and sequential circuits and have investigated the impact of the netlist size on Pythia's performance.

Chapter 5

ZK-SHERLOCK: EXPOSING HARDWARE TROJANS IN ZERO-KNOWLEDGE

As integrated circuit (IC) design and manufacturing have become highly globalized, hardware security risks become more prominent as malicious parties can exploit multiple stages of the supply chain for profit. Two potential targets in this chain are third-party intellectual property (3PIP) vendors and their customers. Untrusted parties can insert hardware Trojans into 3PIP circuit designs that can both alter device functionalities when triggered or create a side channel to leak sensitive information such as cryptographic keys. To mitigate this risk, the absence of Trojans in 3PIP designs should be verified before integration, imposing a major challenge for vendors who have to argue their IPs are safe to use, while also maintaining the privacy of their designs before ownership is transferred. To achieve this goal, in this work we employ modern cryptographic protocols for *zero-knowledge proofs* and enable 3PIP vendors prove an IP design is free of Trojan triggers without disclosing the corresponding netlist. Our approach uses a specialized circuit compiler that transforms arbitrary netlists into a zero-knowledge-friendly format and introduces a versatile Trojan detection module that maintains the privacy of the actual netlist.

5.1 Introduction

Integrated Circuit (IC) designs are embedded in most electronic equipment, and as a result, IC security has become of crucial importance as the globalized economy heavily relies on System-on-Chip (SoC) designs. The IC supply chain depends on procuring a variety of Intellectual Property (IP) cores from third-party vendors (3PIP) and integrating them with components that are designed in-house to fabricate the

IC [203]. However, the integrity of these externally developed IPs cannot always be guaranteed, and as a result, malicious actors can potentially inject hardware Trojans to the SoC designs. Such malicious modifications in hardware could be triggered under certain conditions (e.g., user input, time-based, etc.) or be always on [146]. Rarely-activated Trojans are typically programmed to engage only under a unique set of circumstances created by an attacker and are hard to detect when in their dormant state [224]. When activated, Trojans can alter device functionality by influencing output wires or creating a side channel through which sensitive data can be leaked. It is critical for the IC supply chain to address security concerns, such as hardware Trojans, instead of solely on functionality and runtime performance.

IP core verification is a crucial step of SoC design [95], during which IP consumers provide functional requirements to the vendors and the 3PIP vendors design circuits that meet these specifications.[1] The goal of IP core verification is to convince system integrators about the functionality of the generated 3PIP designs. Thus, ensuring that the circuit is compliant with the specified constraints while achieving a high degree of testability is a crucial consideration in the IC supply chain. Most common solutions include formal logic verification [141], simulation-based methods [177], and application-specific instruction-set processors [221]. Previous solutions are mostly geared toward IP *functional* verification, but fail to protect the privacy of the IP designs. However, recent efforts have also focused on using cryptographic protocols such as homomorphic encryption and zero-knowledge proofs (ZKP) to enhance the security of transactions in the IC supply chain. [125, 154] securely outsource the evaluation of 3PIP netlists to third parties to ensure the confidentiality of the circuit inputs, however, the actual netlist is still visible since homomorphic encryption does not provide functional privacy (only data privacy). [178, 182] preserve the privacy of the netlist using ZKPs but only focus on functional verification and fail to address the increasingly important issue of hardware Trojans. Indeed, existing solutions offer no support for

[1] We use the terms IP consumer and system integrator interchangeably.

system integrators to confirm that the IP they are purchasing does not contain any malicious modifications, without inspecting it themselves.

There are two classes of defenses against hardware Trojans that both require access to the IP, namely invasive and non-invasive [203]. The former incurs significant costs as it requires expensive equipment and renders the IP unusable afterwards; the latter relies on functional and statistical IC testing, such as path-delay measurements [140], and gate-level characterization [4]. Such defenses require unrestricted access to the IP, as well as the statistical distribution of gate characteristics.

In this work, we propose zk-Sherlock, a novel framework for detecting hardware Trojans in zero-knowledge (ZK), i.e., *without allowing access to the circuit*. Our methodology introduces a custom ZK-friendly algorithm for Trojan detection that resolves the deadlock between 3PIP vendors and IP customers. This deadlock is created by the mutual distrust between vendors and consumers: vendors may withhold an IP before receiving payment from customers to avoid the risk of IP theft, while customers may refuse to purchase an IP until they are convinced that it satisfies their requirements. A key contribution of zk-Sherlock is the translation of a netlist into a ZK-friendly format that enables testing using public input vectors to detect any *gates with the least switching activity* and argue about the presence of potentially malicious logic. Our main observation is that the majority of logic gates in a netlist would switch for most input pairs. zk-Sherlock leverages this observation to tally the total number of switched gates across multiple evaluations with different inputs and detect the ones that have not switched. We evaluate our approach using multiple benchmarks from ISCAS '85 and '89 [59,60] with judiciously injected Trojans following the methodology of [86] (as in the TRIT benchmarks on Trust-Hub [206]).

5.2 Zero-Knowledge Trojan Detection

5.2.1 Threat Model

We assume threats in the IC supply chain, from the design to IP integration.

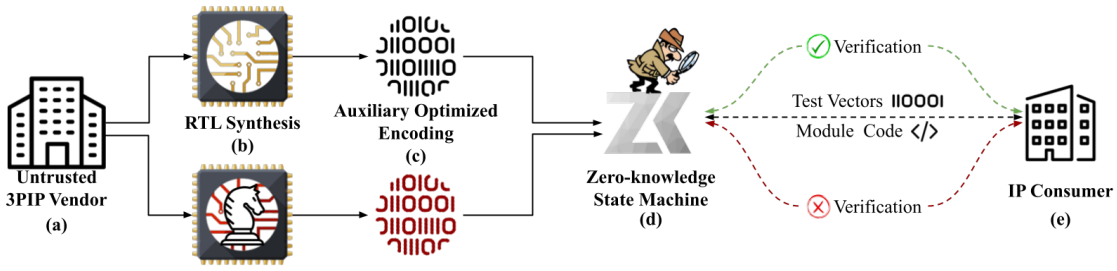


Figure 5.1: Overview of zk-Sherlock. (a) $\mathcal{P}$ possesses an IP described in a Hardware Description Language that has some agreed-upon functional specifications. (b) The 3PIP vendor ($\mathcal{P}$) synthesizes the IP and generates a gate-level netlist, determining the correct evaluation order of the gates. (c) The 3PIP vendor transforms the IP into a ZK-friendly encoding for the Trojan detection state-machine $\mathbb{SM}$. (d) $\mathcal{P}$ executes $\mathbb{SM}$ using the netlist as private input and public test vectors chosen by $\mathcal{V}$. (e) The two parties interact and $\mathcal{P}$ convinces $\mathcal{V}$ that the IP is Trojan-free and that $\mathbb{SM}$ was evaluated correctly.

Cheating $\mathcal{P}$. A cheating $\mathcal{P}$ (i.e., the IP vendor) has financial incentives to deceive the consumer by falsely claiming that they possess an IP with certain functional specifications while the IP could be embedded with malicious circuitry at certain locations. In one scenario, $\mathcal{P}$ may try to deceive an honest $\mathcal{V}$ by trying to sell an IP with a Trojan that alters the agreed-upon functionality. In another case, $\mathcal{P}$ has performed malicious gate modifications to the IP while still meeting the agreed-upon functionality. The system integrator (i.e., buyer) has to test the IP with multiple input vectors, and also verify the correctness of the ZKP. $\mathcal{P}$ succeeds if they can break the soundness of the protocol with probability greater than $2^{-\lambda}$ (where λ is zk-Sherlock's security parameter) by producing a fake ZKP that will convince an honest $\mathcal{V}$ to accept it.

Cheating $\mathcal{V}$. We assume a cheating $\mathcal{V}$ (i.e., an IP consumer) that follows the protocol but also have incentives to extract information about the private IP from an honest $\mathcal{P}$. More specifically, $\mathcal{V}$ may attempt to extract and learn the IP netlist before paying, even though the vendor wants to keep that netlist secret until payment is received. In other words, $\mathcal{V}$ wants to break the *zero-knowledge* property of the protocol and learn the private IP $\mathtt{w}$, which is infeasible if $\mathcal{P}$ follows the protocol faithfully. Notably, $\mathcal{V}$ can only learn that $\mathbb{SM}(\mathtt{x}, \mathtt{w}) = \mathtt{y}$ (in our case $\mathbb{SM}$ checks if the IP $\mathtt{w}$ is Trojan-free).

5.2.2 Overview of our Methodology

In this work, we present zk-Sherlock, a novel methodology for privacy-preserving hardware Trojan detection. Our approach enables 3PIP vendors to prove to system integrators that: a) they possess an IP with some predefined functional specifications, and b) the IP is Trojan-free without revealing anything about its netlist. zk-Sherlock utilizes zero-knowledge proofs to detect any hardware Trojan triggers by inspecting the switching activity of every gate and identifying any non-switching logic *without sharing the netlist* with the IP consumer. To that end, we have designed a specialized ZK state machine $\mathbb{SM}$ that consumes a netlist as private input $\mathtt{w}$ and proves that the circuit has not been embedded with malicious logic. More specifically, $\mathbb{SM}$ is a public *circuit simulator* that evaluates gates and records their switching activity for different input/output pairs. Most gates in a netlist will flip after relatively few input sets unless these gates trigger a rarely-activated Trojan. Thus, our observation is that after evaluating a small number of possible input pairs with zk-Sherlock, all gates should have switched (with high probability), except for Trojan logic. zk-Sherlock then offers provable guarantees on the computational integrity of $\mathbb{SM}$, i.e., that the circuit simulation was performed faithfully. A high-level overview of zk-Sherlock is depicted in Fig. 5.1 and discussed in the following paragraphs.

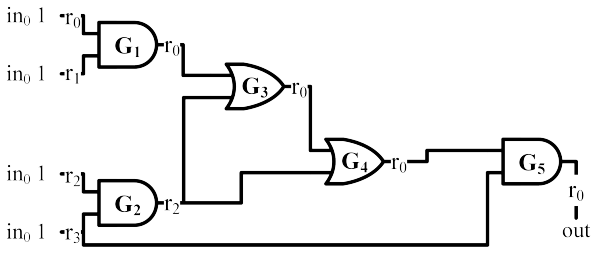
Privacy-Preserving Functional Verification. To solve the mutual distrust between 3PIP vendors ($\mathcal{P}$) and IP consumers ($\mathcal{V}$), zk-Sherlock first needs to prove that a secret IP adheres to some predetermined specifications. As shown in Fig. 5.1(a), the untrusted 3PIP vendor synthesizes an IP described in a Hardware Description Language (HDL),[2] creates a gate-level netlist, and then uses the specialized compiler of zk-Sherlock to transform it into a ZK-friendly encoding, as shown in Fig. 5.1(c). More specifically, our compiler guarantees that this encoding can be evaluated sequentially (i.e., one gate at a time) and does not have any inter-dependencies between the gate inputs and the outputs from previous gates, thus it can be evaluated by zk-Sherlock's

[2] Without loss of generality, zk-Sherlock uses Verilog.

ZK state machine. During the third step (in Fig. 5.1(c)), we apply various optimization techniques to minimize the total number of intermediate wires by employing graph-coloring techniques. Next, $\mathcal{P}$ and $\mathcal{V}$ agree on the functional simulation algorithm $\mathbb{SM}$, shown in (d), and the latter provides public input vectors $\mathbb{x}$. Finally, the 3PIP vendor runs $\mathbb{SM}$ locally to simulate the private netlist $\mathbb{w}$ with inputs $\mathbb{x}$ and computes a public circuit output $\mathbb{y}$, which is then checked by $\mathcal{V}$ along with the cryptographic proof that every step of $\mathbb{SM}$ was executed faithfully.

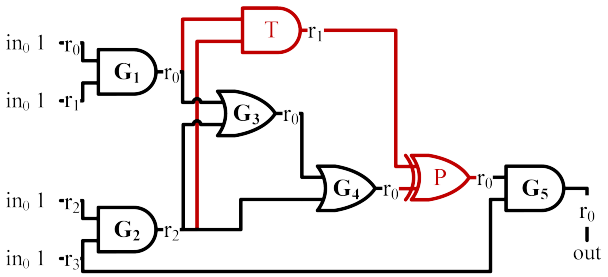
Privacy-Preserving Trojan Detection. After proving that the IP adheres to some agreed-upon functional specifications, zk-Sherlock needs to prove that the netlist does not contain any Trojan trigger logic. As malicious parties aim to leak sensitive information or induce errors by inserting Trojans in circuits that are triggered under special conditions, the gates that comprise the injected Trojan logic are *nearly-unused* and are *rarely activated* on common inputs. Towards that end, we have expanded the state machine $\mathbb{SM}$ to also analyze the gate-switching activity (i.e., when the output signal of the gate flips) of all the gates in the circuit across multiple inputs in order to detect malicious triggers. An important contribution of our work is that $\mathbb{SM}$ acts both as a circuit simulator (to test the netlist functionality in ZK), and as a gate-switching activity analyzer at the same time, combining functional verification and Trojan detection. The circuit compiler of zk-Sherlock translates netlist gates into a sequence of $\mathbb{SM}$ instructions that track which gates have low switching activity over different input pairs and flags them as potentially malicious. Contrary to existing hardware Trojan detection mechanisms, our ZKP approach hides the IP netlist from the verifier.

Fig. 5.2 demonstrates how a hardware Trojan can affect the functionality of a circuit under only a single input combination. To clarify how this encoding is evaluated, we show a two-dimensional table on the right-hand side of both (a) and (b) that represent four $\mathbb{SM}$ registers ($r_0 - r_3$) and how the values in these registers change after the evaluation of each gate. For example, all registers are initialized with "1" and after the evaluation of G_1, its output "1" is written to r_0 (underlined in the table). In Fig. 5.2 (b), we observe that the trigger T of the Trojan is only activated when all inputs are



	r0	r1	r2	r3
G1	*1*	1	1	1
G2	1	1	*1*	1
G3	*1*	1	1	1
G4	*1*	1	1	1
G5	*1*	1	1	1
Out	*1*	1	1	1

(a) Trojan-free circuit.



	r0	r1	r2	r3
G1	*1*	1	1	1
G2	1	1	*1*	1
T	1	*1*	1	1
G3	*1*	1	1	1
G4	*1*	1	1	1
P	*1*	1	1	1
G5	*0*	1	1	1
Out	*0*	1	1	1

(b) Circuit embedded with a Trojan.

Figure 5.2: The gates are labeled by the evaluation order (first G_1, then G_2, etc.) and are also shown on the rows of the tables (execution trace). The underlined values in the tables show which simulation variable was overwritten after the evaluation of the gate. The variables $r0 - r3$ represent four $\mathbb{SM}$ registers, also shown at the outputs of the gates. (a) shows a circuit that outputs "1" when all four inputs are set to high (*note:* there exist more combinations to output "1"). (b) shows the same circuit as (a) after being injected with an example Trojan that is only activated when all inputs are set to "1".

"1", which only happens in one out of 2^4 different input combinations, rendering gate T the gate with the rarest switching activity. However, we do not observe similar behavior for the payload gate P, which switches for various inputs (e.g., 1100). The intuition of zk-Sherlock is motivated by the aforementioned observation, i.e., determine the gates with a suspiciously low switching activity as they can potentially be part of a Trojan trigger.

zk-Sherlock back-end. zk-Sherlock utilizes the Zilch framework [184] as the cryptography back-end to argue about the correctness of $\mathbb{SM}$. Internally, zk-Sherlock leverages the MIPS-like assembly programming language of Zilch to implement the state

machine, which simulates the evaluation of a circuit and computes all gate-switching activity. At a technical level, Zilch enforces different cryptographic constraints that should hold during each $\mathbb{SM}$ transition and are used to prove the correctness of the execution of $\mathbb{SM}$ in zero knowledge.

The initial state of the zk-Sherlock state machine is filled with multiple blocks, each with 64 1-bit registers initialized to zero. With every gate evaluation, $\mathbb{SM}$ modifies at most one register and copies all the blocks into a new state. The sequence of states forms a two-dimensional table that represents the *execution trace* of the $\mathbb{SM}$ (Fig. 5.2). The type and the index of the update on the $\mathbb{SM}$ state correspond to the different operation that is performed in ZK. For instance, an arithmetic operation (e.g., addition) will modify the state in a different way than a bitwise operation (e.g., `AND`, `OR`).

Our state machine consumes *private* inputs that correspond to the netlist description (known only to $\mathcal{P}$) and *public* test vector inputs that $\mathcal{V}$ provides. Utilizing the Zilch back-end, zk-Sherlock cryptographically asserts that all state machine transitions were performed in accordance to the operation in the encoded netlist $\mathtt{w}$; this is enforced using polynomial constraints over the transitions that assert their satisfiability to $\mathcal{V}$. Effectively, zk-Sherlock convinces the IP consumer that a secret netlist has certain functional specifications and that it is Trojan-free without revealing its composition.

5.2.3 Serialized Encoding for State Machine

To generate a proof, zk-Sherlock synthesizes HDL programs into netlists consisting of Boolean gates and flip-flops. Our approach utilizes the Yosys Open SYnthesis Suite for RTL synthesis to generate Electronic Design Interchange Format (EDIF) netlists based on Verilog files [244]. We optimize the netlist during RTL synthesis with Yosys by converting the entire circuit into standard two-input logic gates and removing unused wires. Next, zk-Sherlock associates each logic gate with specific input and output wires (as shown in Fig. 5.2) with $\mathbb{SM}$ registers (e.g., $r_0 - r_3$). An important step for our compiler is to identify gate dependencies by creating a directed acyclic graph, running a topological sort to eliminate dependencies, and finally assigning unique $\mathbb{SM}$

registers to the wires. For instance, in order to evaluate the gate G_3 in Fig. 5.2(a), gates G_1 and G_2 have to be evaluated first. Notably, our compiler applies an additional optimization to further reduce the total number of registers used, as the state size (i.e., the total number of registers in $\mathbb{SM}$) can impact the execution time of our ZK back-end. Finally, the zk-Sherlock compiler serializes the circuit as a sequence of MIPS-like instructions for Zilch, which is ultimately passed as the private input $\mathtt{w}$ to $\mathbb{SM}$, which evaluates the serialized netlist for a given test vector.

The approach depicted above is directly applicable to combinational circuits, but sequential circuits need additional considerations. Since sequential circuits require more than one clock cycle to evaluate completely, we need multiple iterations over a given netlist. Thus, we unroll the circuit for the desired number of clock cycles and when a flip-flop is encountered during evaluation, we propagate its input signal to its output signal at the next clock cycle. While this approach correctly emulates sequential circuits, it is suboptimal when used with our zk-Sherlock module. Often, certain combinational logic segments do not change between clock cycles and do not need to be re-evaluated every time. Therefore, we omit gates that are not connected to an upstream flip-flop to account for this and avoid redundant computations.

Without loss of generality, Fig. 5.3 shows the serialized encoding for an IP with 64 gates (we only show the first, second, and last gate for simplicity) and how zk-Sherlock tracks the gate switching activity (shown as "Switching Gates") across different iterations. Our encoding consists of: gate type (e.g., `AND`), unique gate identifier (depends on the number of gates in the IP), register block ID#, and offset inside the register block for the two input wires and the output wire. The state encoding is organized into multiple blocks (each block holds 64 bits), and each bit represents one $\mathbb{SM}$ register. For instance, the first line of the encoding reads the 10^{th} and the 12^{th} bits of the first block as inputs and writes at the 13^{th} bit of the first 64-bit block. For simplicity, in Fig. 5.3 we only assume two register blocks: the "Switching Gates" block tracks the gate identifiers that have switched, while the "Gate Outputs" block stores the gate outputs. At the end of the first cycle simulation, some gates have switched and

this is reflected in the bottom left block, along with gate outputs in this cycle (bottom right). As more cycles are evaluated, the "Switching Gates" blocks will continue to reflect the number of gates that have encountered a state change. After a specified number of iterations, the presence of a zero in this block suggests a potential Trojan trigger. The total number of iterations is decided by $\mathcal{V}$, so that more iterations offer increased assurance against Trojans.

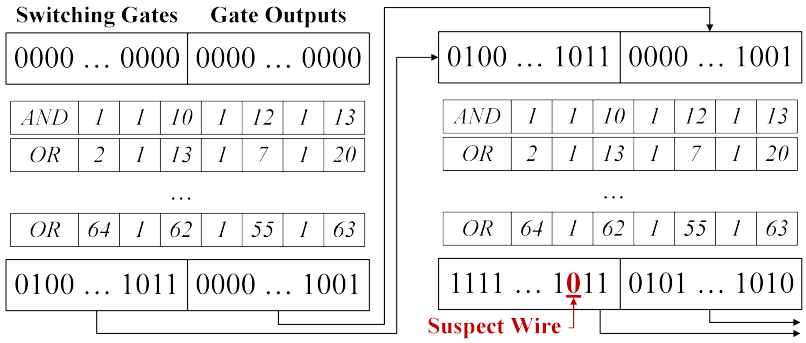


Figure 5.3: Abstraction of two cycles of an 64-gate circuit in zk-Sherlock. "Switching Gates" block and "Gate Outputs" block keep track of the switched gates and the gate outputs, respectively.

5.2.4 State Machine Evaluation

Output generation. We simulate the circuit as follows: First, zk-Sherlock reads a Boolean gate from $\mathtt{w}$ and performs the corresponding Boolean operation on the two input wires to compute the correct output. All gates with no dependencies (i.e., first layer gates) get inputs directly from the public input $\mathtt{x}$, while all intermediate and output gates read their inputs from the outputs of preceding gates which are encoded in different registers in the $\mathbb{SM}$. Our compiler guarantees that the input and output $\mathbb{SM}$ registers map to the correct gates. This way, $\mathbb{SM}$ can keep track of the identifiers of the specific gates that have switched. Internally, we use multiple register blocks (depicted in Fig. 5.3 as "Switching Gates"), where each register corresponds to the switching activity of a particular gate.

Computational Integrity. To prevent a malicious $\mathcal{P}$ from deceiving an honest $\mathcal{V}$ by switching the IP under test across different executions, zk-Sherlock applies a secure

hash function to compute two secure digests: one for the IP netlist itself and one for the state registers. Then, at the beginning of each new cycle, $\mathcal{P}$ initializes the $\mathbb{SM}$ registers with the previous execution state and computes the secure hash of the new registers to prove that she propagated the state from the previous cycle. $\mathcal{P}$ evaluates the circuit again, keeping track of which gates switched during the current and all previous executions. At the same time, $\mathcal{P}$ recomputes the hash of the private IP and proves to $\mathcal{V}$ that the same $\mathbb{w}$ was used across all the executions. Notably, $\mathcal{V}$ does not learn any intermediate results about the gate switching activity as the intermediate registers are stored at the beginning of $\mathbb{w}$.

5.3 Experimental Results

We evaluate our methodology using selected benchmarks from the ISCAS'85 and ISCAS'89 suites [59, 60]. During synthesis, we used the `proc`, `flatten`, `synth`, and `abc -g simple` flags in Yosys to generate circuits with standard 2-input logic gates. Moreover, our zk-Sherlock compiler is implemented in Python 3, while the state machine simulator is implemented in a MIPS-like assembly language for ZKPs. As the back-end of zk-Sherlock can benefit from multiple threads to accelerate the proving time, we used an m5.24xlarge AWS EC2 instance featuring two Intel Xeon Platinum 8175M processors at 2.5 GHz.

In a realistic scenario, zk-Sherlock uses multiple input test vectors chosen by $\mathcal{V}$, which correspond to the functional properties the IP consumer wants to assert. The input vector values do not impact the proving time, but affect how the gates switch. Because $\mathcal{V}$ has no knowledge of the underlying circuit, the best method for choosing input vectors is to generate them randomly. As shown in our experiments, random test vectors cause all the gates in a circuit to flip relatively quickly with a high probability when a Trojan is not present.

Fig. 5.4 shows the amortized execution time for both the prover and verifier for a set of inputs. In the case of sequential circuits (from the ISCAS'89), the presented time reflects the cost per cycle. In practice, multiple input vectors should be utilized

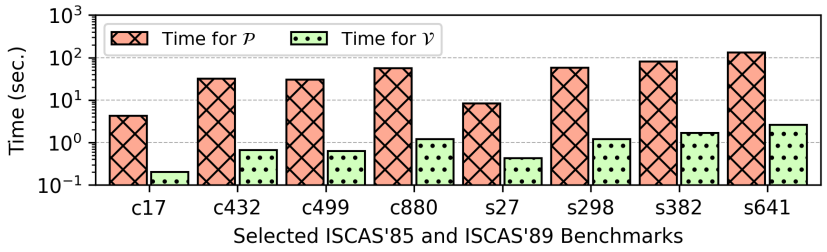


Figure 5.4: Experimental timings for $\mathcal{P}$ and $\mathcal{V}$ per input-pair for selected benchmarks.

at the discretion of $\mathcal{V}$ to minimize the risk of false positives (where one or more non-malicious gates have yet to switch across all input sets provided). The timings for $\mathcal{P}$ scale linearly with the number of logic gates per circuit and the execution time for $\mathcal{V}$ scales logarithmically with increasing circuit sizes. In addition, the size of the execution trace scales with the number of $\mathbb{SM}$ registers required to simulate the circuit, which can impact the execution time. Generally, the wider the circuit, the more register blocks are required; for instance, c17 requires 5 intermediate wires at its widest point, which can fit into a single 64-bit packed block, while c880 requires 87 registers that fit into two 64-bit blocks. Notably, the proving times for all circuits are independent of the input pattern, as an identical set of gates is visited for each run.

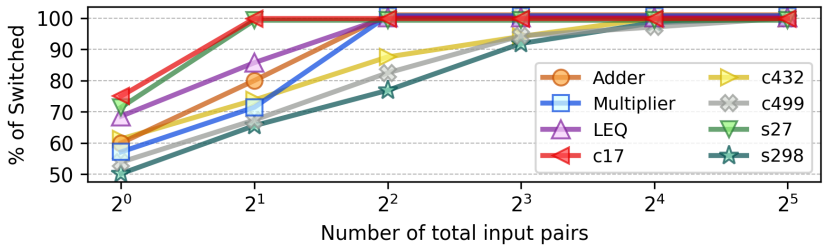


Figure 5.5: Percentage of the total gates that switched over increasing number of input pairs for selected Trojan-free benchmarks.

The verifier needs to choose enough test vectors to avoid false positives. While each circuit's gate switching activity will vary, in Fig. 5.5 we investigate the percentage of possible input pairs required to flip all non-malicious gates in an assortment of eight circuits with random input wire sets. We found that 32 input sets were sufficient to flip all gates, and hence resulted in no false positives, in all eight Trojan-free circuits,

while all but one circuit had flipped earlier. However, random inputs assume that $\mathcal{V}$ has no knowledge about the *behavior* of the circuit, whereas practically, the prospective buyer of an IP is aware of the expected functionality. Thus, $\mathcal{V}$ can cleverly choose pairs to cover a large number of cases in a small number of input pairs; for instance, in a simple example with a 2-bit less-than-or-equal circuit, $\mathcal{V}$ can choose two sets of inputs that will cause all of the gates to switch. If a Trojan is hidden in such a circuit, our approach can detect it with just 2 input pairs.

To further assess our approach, we injected Trojans in c17, c432, c499, s27, and s298 ISCAS circuits following the methodology of TRIT [86].[3] We carefully modified the benchmarks so that the Trojans get triggered with only a rare combination of the input wires and alter specific output wires, similarly to Fig. 5.2. zk-Sherlock successfully detected the Trojans in all cases; interestingly, we note that c499, the largest combinational circuit tested, yielded the lowest probability of false negatives for a given number of inputs, as there are 41 input wires and hence 2^{41} possible inputs and only 32 inputs were required to flip all benign wires. Therefore, since the Trojan can only trigger on a very rare input combination, the chance of the Trojan flipping during the tested input sets is approximately zero. For smaller circuits, such as c17 and s27, there are far fewer possible input pairs (32 and 128 respectively), so the probability of false negative increases to approximately 6% for c17 and 1.5% for s27 as each requires 2 input pairs to successfully flip all benign wires.

5.4 Related Work

zk-Sherlock focuses on hardware Trojans that are activated based on particular user inputs and thus the trigger nodes are rarely executed. The authors of [192] use an advised genetic algorithm to generate test vectors to detect Trojans based on rare nodes. Similarly, FANCI [240] performs functional analysis to flag logic that is unlikely to affect the circuit outputs, whereas VeriTrust [250] identifies potential Trojan wires

[3] Precompiled TRIT benchmarks from Trust-Hub [206] are incompatible with Yosys so we created equivalent benchmarks based on their ISCAS circuits.

by examining verification corner cases. All of these techniques assume that the circuit design is available for inspection and that details such as the statistical distribution of gate characteristics are known. A critical benefit of zk-Sherlock is that it proves to the IP consumer that a netlist is Trojan-free without revealing anything about it.

Pythia [178, 182] describes an approach related to zk-Sherlock by introducing the problem of functional IP verification in zero knowledge. However, these works focus on ensuring that the IP has some functional properties and that it also satisfies constraints related to area, performance, and power consumption by checking different input-output pairs provided by IP consumers. Orthogonal to these earlier works, our approach transforms the problem of IP verification to a zero-knowledge protocol and introduces a powerful encoding and a versatile $\mathbb{SM}$ that allows tracking the switching activity to offer assurance about hardware Trojan triggers.

5.5 Concluding Remarks

In this paper, we have proposed a unique methodology for detecting hardware Trojans in zero knowledge (i.e., without having access to the IP netlist). zk-Sherlock proposes a new encoding that enables evaluating both the circuit and computing the gate-switching activity at the same time. Using this gate activity, our methodology identifies the nodes that are triggered under rare conditions and thus flags them as potentially malicious logic. In effect, zk-Sherlock enables 3PIP vendors to convince system integrators that a netlist is Trojan-free so that integrators have only black-box access to the IP by submitting test vectors. Additionally, as the system integrator may provide inputs, zk-Sherlock employs a secure hashing to confirm that across different executions: (a) the same secret netlist was used, and (b) the gate-switching activity counters were propagated correctly. Our experiments with ISCAS'85 and '89 benchmarks demonstrate that our approach converges quickly to "Suspected Trojan" or "Trojan-free" classification for an IP under test.

Chapter 6

MASQUERADE: VERIFIABLE MULTI-PARTY AGGREGATION WITH SECURE MULTIPLICATIVE COMMITMENTS

6.1 Introduction

A variety of applications require gathering and aggregating data from different organizations or individuals to perform studies, collect statistics, and mine interesting patterns about the participating population. Common applications of data aggregation can be found in finance, where financial institutions need to gain insights on customer clusters, as well as in healthcare, where aggregated data are used to discover effective treatments and track the spread of highly infectious diseases. In certain use cases, all participants trust the entity conducting the study (called *data curator*) with their personal data, which results in a straightforward solution. Nevertheless, in the case of a *curious* curator that has incentives to peak into sensitive user data (e.g., for targeted advertisements or even affect election results [135]), the problem becomes significantly more challenging to both protect the privacy of each participant and compute the desired statistics.

Common approaches to support privacy-preserving crowdsourcing rely on adding noise to the collected data so that individual inputs cannot be deduced from the output. Unfortunately, these techniques have limitations since data anonymization can be bypassed [85, 189], while local differential privacy (DP) may generate excessive accumulated noise. A practical alternative entails adding noise to the output of the queries using DP techniques so that the presence or absence of a single user cannot be detected from the query answers [98]. Unfortunately, such approaches assume a trusted curator and mostly focus on output privacy, i.e., they do not protect the data privacy of each individual user from the curator. The works in [42, 103, 214] provide strong privacy

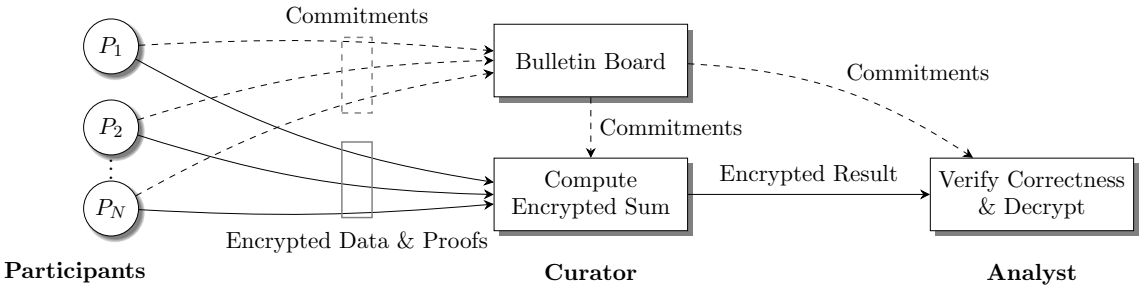


Figure 6.1: Overview of Masquerade. Each participant sends their encrypted data along with a zero-knowledge proof that their ciphertext is well-formed to the curator, who in turn performs the homomorphic aggregation. Participants also publish their commitments on a bulletin board so that everyone can access them and verify the correctness of the encrypted sum. Finally, the analyst decrypts and publishes the result of the computation.

guarantees and high utility, however, they still add a non-negligible amount of noise to the results.

We present a novel construction for *private data aggregation (PDA)* [214], a special case of secure multiparty computation (MPC) in which multiple participants jointly compute a function over their private data while keeping them private [120, 246]. MPC assumes that client data are distributed amongst two or more computing parties and that the data are secure as long as these parties do not collude. While it is possible to use general-purpose MPC to collect statistics from multiple individuals, however, this comes with significant overheads due to support for arbitrary computation. Additionally, many MPC frameworks [6, 148, 166] allow only the computing parties to provide *private inputs*; at the same time, since the input parties are allowed to send *any input* to the aggregation, it is possible for a malicious participant to poison the result by uploading invalid data. To defend against such attacks, specialized checks have to be implemented within MPC that require additional communication and computation. Finally, generic solutions do not provide public verifiability since data confidentiality relies on the computing parties not publishing their shares.

Homomorphic encryption (HE) allows performing meaningful calculations directly on encrypted data without decrypting it, which enables outsourcing a computation to semi-honest servers and returning the encrypted result to the party that initiated the computation [114,174,229]. In particular, HE offers an elegant solution for PDA: each participant encrypts and uploads their private data to a computing party (curator) that performs the aggregation, which in turn returns the encrypted sum to the requesting party (analyst), who decrypts the final sum. Partially HE (PHE) schemes support only one mathematical operation (either addition or multiplication) to be performed on ciphertexts, whereas fully HE (FHE) enables both, but incurs significant overheads, especially for non-linear operations and comparisons. Nevertheless, HE alone does not offer any integrity guarantees to the public on the correctness of the computation. Therefore, the public has to rely on the assumption that all computations are error-free and no party has introduced an invalid ciphertext to the homomorphic sum. Without special integrity checks to enable public audits, relying solely on HE allows computation errors to remain undetected.

Distributed ledgers (e.g., blockchains) can be used as public bulletin boards to enhance transparency and enable public verifiability. However, most of them are either public and reveal data patterns (e.g., Bitcoin [187]), or private without support for public verifiability. The Zerocash protocol [32] builds an anonymous cryptocurrency on top of Bitcoin by utilizing zk-SNARKs [33]. zk-SNARKs are non-interactive proofs that are used to hide the transaction amounts, the senders, and the receivers and still prove the integrity of the transactions. Unfortunately, zk-SNARKs require an expensive pre-processing phase to be carried out by a trusted party to set up the system and destroy any randomness used. If this randomness is leaked, any adversary can forge false proofs and break the protocol's soundness.

Our contribution. This work introduces *Masquerade*, a novel framework for computing statistics on private data by allowing multiple users to share their data points with an *oblivious curator*, who is unable to deduce any information from the data (other than what it is learned from the published result). In particular, Masquerade involves

one *analyst* (i.e., a party that originates the computation and will eventually reveal the statistics), P *participants* that share their private data, and the curator who gathers the encrypted data and performs the PDA. The participants also publish commitments to their data to a public bulletin board, which can be used to verify the correctness of the result by any auditor. Masquerade introduces a new multiplicative-homomorphic commitment scheme that does not rely on any trusted party.

An overview of Masquerade is illustrated in Fig. 6.1: The analyst outsources the PDA to the curator and verifies the correctness of the results (we remark that Masquerade enables public audits for detecting homomorphic computation errors). The curator only learns the homomorphic ciphertexts of the participants' messages, which do not reveal anything about their sensitive data. Similarly, the curator shares with the analyst only the encrypted result of the computation, rendering it impossible for the analyst to learn anything except what can be inferred from the final output. Each participant publishes commitments to their encrypted messages to a public ledger, and the analyst combines them homomorphically to attest to the correctness of the encrypted result (e.g., detect double-voting). Notably, our methodology addresses the problem of *malicious participants attempting to tamper with the aggregation* by leveraging two *non-interactive* zero-knowledge proof (ZKP) protocols. These protocols ensure that the participants follow the protocol faithfully by proving to the curator that their encrypted data points lie either in a range or within a set of valid messages, without disclosing the actual plaintexts. Therefore, malicious participants cannot corrupt the homomorphic sum by sending encryptions of invalid messages.

Masquerade enables two classes of studies: *Quantitative* and *Categorical*. An example of the former class includes privacy-preserving smart metering, where users share their electricity consumption readings with a service provider over consecutive time periods to calculate their bill. Smart metering imposes a potential risk to user privacy since fine-grained readings may disclose sensitive information about the clients' habits. Examples in the latter class include surveys where individuals select one category among multiple options. For instance, a study that investigates if the juries in

federal courts come from a diverse group of socioeconomic status includes sensitive personal information. This application can be instantiated natively using Masquerade, where each jury provides a private categorical input representing their ethnicity. Then, Masquerade privately generates a histogram that represents how diverse a group of juries is. Our scheme establishes a novel way for quantitative and categorical studies by protecting each participant's privacy.

Overall, our contributions can be summarized as follows:

- Design of a novel commitment scheme that enables *homomorphic multiplication* between commitments. We leverage this scheme to generate a global commitment over the encryption of the aggregation result using the individual commitments of the participants and provide *verifiable guarantees* to the public that there was no error in the homomorphic aggregation of the ciphertexts.
- Construct a privacy-preserving data aggregation protocol that is robust against client dropouts and malicious participants who attempt to tamper with the aggregation result.
- A data-encoding methodology to allow homomorphic aggregations for both quantitative variables (i.e., data that take numerical values) and categorical data (i.e., data grouped into discrete classes), which enables a broad range of privacy-preserving studies.

6.2 Our Problem Statement

6.2.1 Overview

Gathering and aggregating data from multiple parties enables powerful analytics. Companies can collect meaningful information about their clients' behavior and demographics, while healthcare researchers can discover more effective treatment methods to accelerate the overall healthcare facility. Such studies correspond to one of two categories based on the type of data being used: The first class includes variables that represent quantities (i.e., *quantitative data*) that take numerical values (such as age and profit). The variables in the second class can take one value from a fixed set of

options and are called *categorical data* (such as age group, nationality, and employment status). In a quantitative study, the curator applies a function f directly to the participants' data (e.g., compute the average age of consumers or a weighted arithmetic mean based on electricity cost), while in a categorical study, the curator applies f in each different category to generate a histogram (e.g., find the total number of consumers by nationality). The latter can be depicted as a bar graph that shows the frequency distribution of the participants' responses.

Private Data Aggregation (PDA) computes similar aggregations and frequencies while tolerating a curious curator that has incentives to peek at sensitive data points (e.g., to enable targeted advertisements) [214]. PDA enables many useful applications by assuring the participants that their data will only be used towards the agreed analysis and private data cannot be tracked back to each individual; notably, the curator will never access the sensitive data. In this work, we are interested in defining a secure PDA protocol for *both quantitative and categorical studies*. To motivate our case study, we summarize an example from each of the two scenarios.

Categorical analysis. This class enables individuals to take part in surveys where they choose their replies from fixed sets of answers. For example, course evaluations include questions with categorical data, where each response is a one-hot encoding from a set of possible answers (e.g., *"The instructor used the time effectively."*, with four possible responses *a) Strongly Disagree, b) Disagree, c) Agree, d) Strongly Agree* on a *Likert scale* [163]). These evaluations comprise an essential indicator for faculty and departments to get feedback and improve their classes. In course evaluations, it is crucial that there is no link between the participant and their responses so faculty are not biased towards individuals, and most importantly, individuals are more likely to provide more accurate comments. In this case, public verifiability guarantees to the students that their evaluations were counted and also the students can prove that they only submitted one response.

Quantitative analysis. In this category, individuals can privately report numerical

values that will be used to compute a total sum, an average, or a weighted average. A notable example is smart metering, which can be applied to multiple participating households. Smart meters communicate the electrical usage almost in real-time to provide better system monitoring and customer billing than traditional meters. However, there are inherent privacy concerns since fine-grained measurements (e.g., one every 15 minutes) may reveal personal information about the number of people in a household and their activities [70, 90]. Masquerade solves this problem by aggregating data over multiple participating households for a given period that would still be sufficient for smart grid operators to perform enhanced monitoring and optimize prices but at the same time will conceal private information. Public verifiability is essential to convince all the households that the result was computed correctly. Masquerade naturally extends this application by computing the weighted arithmetic mean of the participating households, as electricity rates can vary based on the area. This allows participants and communities to compare their electricity consumption based on their (public) area price and gain useful insights about their electricity rates.

6.2.2 Threat Model

The different applications supported by Masquerade require protection against a variety of threats. Our threat model assumes malicious participants and auditors, and a *mixed model* for the curator and the analyst that comprises the passive-corrupt (i.e., semi-honest) and the fail-corrupt models [107]. Our threat model is aligned with the models of secure two-party computation and it assumes that the curator and the analyst *will not collude*. Instantiating our protocol assuming two parties with mutual distrust enables computing the aggregation without exposing sensitive information about the participants' data [145, 246].

Curator. A *passive-corrupt* curator may have incentives to peek into the participants' sensitive data to extract information about their encrypted messages. In our approach, participants' inputs are protected using a secure probabilistic PHE scheme (our protocol relies on the Paillier cryptosystem with a 2048-bit modulus N). A *fail-corrupt*

curator may stop the communication from the participants at an arbitrary time during the protocol with a result of altering the final sum (e.g., by being offline for a certain period of time and failing to aggregate all the ciphertexts). Under the fail-corrupt model, the curator may introduce non-adversarial or random faults, which could affect the integrity of the computation [207]. In this paper, we borrow the *mixed model* introduced in [107], in which the adversary may passively corrupt and/or fail-corrupt the curator. As PHE does not offer any native integrity protections, we introduce a novel multiplicative homomorphic commitment scheme (section 6.3.1) that provides public verifiability without compromising the participants' privacy. The analyst also employs these public commitments to verify the computational integrity of the homomorphic result.

Participants. We assume *malicious* participants that may attempt to disproportionately affect the analysis by encrypting invalid inputs (e.g., a large negative number). To address this threat, we introduce two non-interactive ZKP protocols that allow the participants to prove to the curator that the provided data encrypts a valid message. The curator verifies the participants' proofs and only considers the private inputs for which the ZKP verification was successful. As soon as the private aggregation is finished, the curator sends to the analyst the list of participants whose proof and commitment were verified successfully to include their commitments for the verification of the encrypted sum. This guarantees that each participant submits a valid input (i.e., within a pre-agreed range) and cannot detect if clients submitted valid but false data (e.g., lie about their age).

Masquerade is also robust against *participant dropouts* who submit their inputs and stop participating. Additionally, to defend against Sybil attacks that create a large number of illicit clients to influence the result [96], both the curator and the analyst agree on a list of identities. Active-adversary attacks can be further thwarted using a Public-Key Infrastructure that issues certificates. Each participant signs their message and the other participants (including the curator and the analyst) can verify the message signatures to prevent forgeries [2]. Finally, to prevent clients from biasing

the result, we allow one private input per client. Our work focuses on protecting data confidentiality, rather than participant anonymity.

Analyst. We assume a *passive-corrupt* and a *fail-corrupt* analyst that is interested in learning the aggregation results but has incentives to extract information about individuals' data and may introduce non-adversarial errors. We consider the bulletin board as an online append-only ledger that is auditable by anyone (e.g., public auditors) and is maintained by the analyst using standard consensus methods [193] or using a public blockchain network (e.g., Ethereum). The analyst posts a proof of correct decryption to the bulletin board providing public verifiability. As long as at least one auditor is honest, our protocol will detect a fail-corrupt analyst.

Auditors. Finally, we allow multiple auditors to inspect the correctness of the PDA protocol and we tolerate all but one to behave maliciously.

6.3 Private Data Aggregation Protocol

In this section, we describe Masquerade, our novel cryptographic construction for secure PDA. Our protocol supports quantitative data *by design* and we extend our construction to support categorical data, as discussed in section 6.3.5. Each participant (a) locally encrypts their private data using Paillier, (b) publishes in a bulletin board a multiplicative commitment on the ciphertext, (c) generates a non-interactive ZKP that enables its recipient to verify that the ciphertext is the encryption of a message that lies in a range of valid messages, and (d) sends their encrypted data along with a proof to the curator. The curator first verifies the validity of the ZKP and the correctness of the commitment, and then homomorphically adds the participant's data to an encrypted accumulator if the proof and the commitment are correct; otherwise, the curator rejects the participant's data. As soon as all participants have sent their data, the curator publishes the encrypted sum.

The analyst audits the protocol by accumulating the homomorphic commitments from the ledger and verifying that they open with the encrypted sum. Since the commitments and the final sum are public, any participant can repeat the same

process and be convinced that their private data were included in the computation, as well as verify the correctness of the encrypted aggregation. We remark that the need for public verifiability remains critical in real-world applications with potentially untrusted participants. Therefore, Masquerade explicitly enables the participants to verify that no errors were introduced in the aggregation result (e.g., by omitted inputs or due to other random faults), since the analyst would not be able to successfully open the commitment otherwise. Upon verifying that the curator's result is correct, the analyst decrypts and publishes the result along with a proof of correct decryption of the final result.

Benefits. A crucial property of our scheme is that it does not require interactive communication between the participants and the curator. Each participant generates and sends encrypted data, the ZKPs, and the commitments locally and then disconnects from the protocol. Since our commitment scheme does not reveal anything about the participants' data, any public ledger that does not rely on a trusted setup is sufficient.

Furthermore, if some clients deliberately stop participating in the protocol (i.e., participant *dropout*), our scheme remains resilient since we do not rely on any user secret for the decryption of the final result. In contrast, in earlier PDA protocols that are based on secret-sharing a single client can easily corrupt the protocol by just withdrawing their participation.

6.3.1 Our Multiplicative Commitment Scheme

Recall from section 2.5 that the Pedersen commitment scheme has additive homomorphism [197]. Here we propose a new commitment scheme based on RSA [202] with multiplicative homomorphism. Let N be a public RSA modulus such that $N = pq$, where p, q are secret primes, let e be a public prime that satisfies $e > N^2$ and $\mathsf{GCD}(e, \phi(N^2)) = 1$, and let g_m be a public element of maximal order in $\mathbb{Z}_{N^2}^{\times}$. To bind to a secret message $m \in \mathbb{Z}_{N^2}^{\times}$, the sender $\mathcal{S}$ generates a random secret $r \in \mathbb{Z}_{N^2}^{\times}$ and calculates c as:

$$c = \mathsf{Com}(m, r) = m^e g_m^r \bmod N^2. \tag{6.1}$$

Assuming that the RSA problem with modulus N^2 and public exponent e is hard to invert, this commitment scheme (intuitively) satisfies the hiding and binding requirements. During the reveal phase, both the secret message m and the randomness r are published, and the verifier checks if they produce the same commitment as c.

Theorem 1. *The scheme with multiplicative homomorphism from Eq. 6.1 satisfies the hiding and binding properties.*

Proof. First, we show that the scheme is perfectly hiding. Sender $\mathcal{S}$ confirms that $N^2 < e$ and e is a prime, so that $\mathsf{GCD}(e, \phi(N^2)) = 1$ and e is invertible $\mod \phi(N^2)$. We will show that $c = m^e g_m^r \bmod N^2$ is indistinguishable from random values: Since g_m is an element of maximal order in $\mathbb{Z}_{N^2}^\times$ (by construction) and r is uniformly random in $\mathbb{Z}_{N^2}^\times$, then $g_m^r \bmod N^2$ is uniformly distributed in a subgroup of $\mathbb{Z}_{N^2}^\times$ with order $|g_m|$. Likewise, the product $m^e g_m^r \bmod N^2$ is also uniformly distributed in a subgroup of $\mathbb{Z}_{N^2}^\times$ with high order $|g_m|$, so it is indistinguishable from a random value; thus, nothing can be learned about m from c.

Next, we prove that our scheme is computationally binding under the RSA assumption (i.e., the problem of computing e^{th} roots modulo a composite N^2 is computationally hard). Our goal is to show that if a $\mathcal{S}$ can find two different messages m, m' with corresponding randomness r, r' that collide to the same commitment $c = c'$, this would imply we can easily compute e^{th} roots modulo N^2 so that the RSA problem would be easy (contradiction). To show this, we assume it is possible to find $m \neq m'$ and $r \neq r'$, so that $c = m^e {g_m}^r = c' = {m'}^e {g_m}^{r'} \bmod N^2$. In this case, ${m^e g_m}^r = {m'^e g_m}^{r'} \bmod N^2 \iff g_m^{r-r'} = (m'/m)^e \bmod N^2$.

Using the fact that $r, r' \in \mathbb{Z}_{N^2}^\times$ and e is a prime larger than N^2, it holds that $r - r' < e$ and also $\mathsf{GCD}(r - r', e) = 1$. Thus, from Bèzout's identity, there exist integers A, B (that can be computed in polynomial time using the extended Euclidean algorithm), so that $A(r - r') + Be = \mathsf{GCD}(r - r', e) = 1$. Then, we have:

$$\begin{aligned} g_m^1 &= g_m^{A(r-r')+Be} = (g_m^{(r-r')})^A g_m^{Be} \bmod N^2 \\ &= ((m'/m)^e)^A (g_m^B)^e \bmod N^2 = (m'/m)^A g_m^B)^e \bmod N^2, \end{aligned}$$

and thus, $g_m^{1/e} = ({}^{m'}/_{m})^A \, g_m^B \bmod N^2$.

The above steps can efficiently compute the e^{th} root of $g_m \bmod N^2$, which contradicts the hardness assumption of RSA for a sufficiently large composite N^2. Therefore, the original assumption that $\mathcal{S}$ can bind a commitment c to more than one message is false, as expected. □

From Eq. 6.1 we observe that our commitment has a multiplicative homomorphic property; we can compute a commitment for the multiplication of two messages m_1 and m_2 by knowing the individual commitments to these two messages. To open this new commitment, the randomness r_1 and r_2 used for each individual commitment should be added. Note that this is distributed identically to a fresh commitment to $m_1 \cdot m_2$. More formally:

$$\begin{aligned}\mathsf{Com}(m_1, r_1) \cdot \mathsf{Com}(m_2, r_2) \bmod N^2 &= ({m_1}^e g_m^{r_1}) \cdot ({m_2}^e g_m^{r_2}) \bmod N^2 \\ &= ((m_1 m_2)^e \cdot g_m^{r_1 + r_2}) \bmod N^2 \\ &= \mathsf{Com}(m_1 m_2, r_1 + r_2),\end{aligned}$$

which can be extended to P messages as follows:

$$\textstyle\prod_{i=1}^{P} \mathsf{Com}(m_i, r_i) = \mathsf{Com}(\prod_{i=1}^{P} m_i, \sum_{i=1}^{P} r_i) \bmod N^2. \tag{6.2}$$

Next, we describe how the analyst leverages this property to generate a commitment over the encrypted sum without having access to the individual encrypted messages, but only to the public commitments of each encrypted message. By the *hiding* property (Theorem 1), each commitment to a message does not reveal anything to the analyst about the message itself. Additionally, the *binding* property guarantees the computational integrity of ciphertext aggregation, as an error in the aggregation result would prevent an auditor from successfully opening the aggregate commitment. Masquerade offers public verifiability by publishing all the commitments to a ledger and enabling any party to audit them.

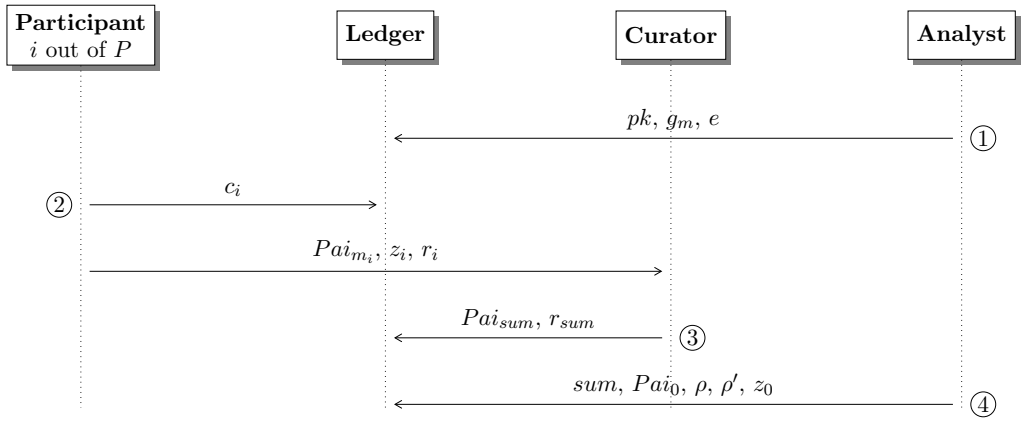


Figure 6.2: The Masquerade Protocol. The numbers 1-4 refer to the algorithms from Fig. 6.3. (1) First, the analyst generates a Paillier key-pair and the public parameters for the commitment scheme and posts the public key including N^2, g_m, and e. (2) Each participant i encrypts their private data m_i and generates a non-interactive ZKP to prove the correctness of ciphertext Pai_{m_i} to the curator. Participants also commit to Pai_{m_i} and publish the commitment values c_i, while they send the random r_i values used for c_i to the curator. (3) Upon verifying the proof and commitment, the curator homomorphically adds Pai_{m_i} to the encrypted aggregation and also adds the r_is. (4) The analyst receives the encrypted sum and the sum of the random r_is from the curator and verifies that the commitment opens successfully. Finally, the analyst creates a non-interactive ZKP that the final result sum is the correct decryption of Pai_{sum}.

6.3.2 Homomorphic Commitments on Homomorphic Data

We now discuss our novel PDA scheme, depicted in Fig. 6.2. First, the analyst generates a Paillier key-pair sk, pk as well as the random prime e and the maximum-order element g required for our commitment scheme (section 6.3.1) and publishes the pk (including modulus N^2), e, and g_m. Notably, we use the same N^2 both as the public *commitment* modulus and as Paillier's *encryption* modulus. Then, each participant probabilistically encrypts their private message using Paillier and commits to the generated ciphertext (recall, these commitments do not reveal anything about either the ciphertext or the plaintext). Then, the participant sends their ciphertext to the curator along with the randomness used for the commitment and publishes the commitment value to the ledger. (We omit ZKPs for now.) The curator calculates the homomorphic addition of the participants' messages, adds the individual commitment randomness r_i, and publishes the Paillier ciphertext along with the sum of all r_i's to

① Key-Generation **(Analyst)**

$sk, pk \xleftarrow{\$} \mathsf{Gen}_{Pai}(\text{security\_level\_in\_bits})$
$N^2 = p^2 \cdot q^2$ using primes $p, q \in sk$
N^2 is used both for Paillier and commitment
Random prime e, s.t. $e > N^2$ and $gcd(e, \phi(N^2)) = 1$
$g_m \xleftarrow{\$}$ Max-order $\mathbb{Z}^*_{N^2}$
Publish to the ledger pk, g_m, e

② Encrypt-Prove-Commit **(Participant)**

$Pai_{m_i} \leftarrow \mathsf{Enc}_{pk}(m_i, \rho_i)$ ▷ $\rho_i \xleftarrow{\$} \mathbb{Z}^*_{N^2}$
$z_i \xleftarrow{\$} \mathcal{P}(Pai_{m_i}, m_i)$ ▷ Prove Pai_{m_i} is an encryption of m_i
$c_i = \mathcal{C}(Pai_{m_i}, r_i)$ ▷ $r_i \xleftarrow{\$} \mathbb{Z}^*_{N^2}$
Publish to the ledger c_i
Privately transmit to Curator Pai_{m_i}, z_i, r_i

③ Aggregate **(Curator)**

Accept/Reject $\leftarrow \mathcal{V}(pk, z_i, Pai_{m_i})$ ▷ Verify the proof
Verify $c_i \stackrel{?}{=} \mathcal{C}(Pai_{m_i}, r_i)$ ▷ Verify the commitment
$Pai_{sum} \leftarrow \prod_{i=1}^{P} Pai_{m_i} \bmod N^2$ ▷ Aggregate ciphertexts
$r_{sum} \leftarrow \sum_{i=1}^{P} r_i$ ▷ Aggregate randomness
Publish to the ledger Pai_{sum}, r_{sum}

④ Decrypt-Audit-Prove **(Analyst)**

Verify $\prod_{i=1}^{P} c_i \bmod N^2 \stackrel{?}{=} \mathcal{C}(Pai_{sum}, r_{sum})$
$sum \leftarrow \mathsf{Dec}_{sk}(Pai_{sum})$ ▷ Decrypt result
$Pai'_{sum} \leftarrow \mathsf{Enc}_{pk}(sum, \rho)$ ▷ $\rho, \rho' \xleftarrow{\$} \mathbb{Z}^*_{N^2}$
$Pai_0 \leftarrow Pai_{sum} * (Pai'_{sum})^{-1}$ ▷ Encryption of zero
$z_0 \xleftarrow{\$} \mathcal{P}(Pai_0, \mathsf{Enc}_{pk}(0, \rho'))$ ▷ Prove $Pai_0 = \mathsf{Enc}_{pk}(0, \rho')$
Publish to the ledger sum, Pai_0, ρ, ρ', z_0

⑤ Public Verification **(Auditor)**

Verify $\prod_{i=1}^{P} c_i \bmod N^2 \stackrel{?}{=} \mathcal{C}(Pai_{sum}, r_{sum})$
Verify $Pai'_{sum} \stackrel{?}{=} \mathsf{Enc}_{pk}(sum, \rho)$ ▷ Verify Pai'_{sum}
Verify $Pai_0 \stackrel{?}{=} Pai_{sum} * (Pai'_{sum})^{-1}$ ▷ Verify Pai_0
Accept/Reject $\leftarrow \mathcal{V}(pk, z_0, Pai_0)$ ▷ Verify the proof

Figure 6.3: The four core algorithms of the Masquerade protocol. In the top right corner (inside the parentheses), we indicate which party runs each algorithm. $\mathcal{P}$, $\mathcal{V}$, and $\mathcal{C}$, stand for the prover, the verifier and the commitment algorithm, respectively. Both $\mathcal{P}$ and $\mathcal{V}$ algorithms are discussed in Section 6.3.4.

the bulletin board. Finally, the analyst verifies the integrity of the homomorphic result using the formula of Eq. 6.2 to ensure the multiplication of the client commitments along with the aggregated randomness equals the commitment over the encrypted sum.

For example, given two participant messages m_1 and m_2, the curator receives two ciphertexts $\mathsf{Enc}_{pk}(m_1)$ and $\mathsf{Enc}_{pk}(m_2)$ and computes $\mathsf{Enc}_{pk}(m_1) \cdot \mathsf{Enc}_{pk}(m_2) \bmod N^2$ (here, we omit the randomness of Paillier encryption). Moreover, the curator sums the *commitments' randomness* $(r_1 + r_2)$. The analyst reads the published commitments to these two ciphertexts and can verify the commitment to the encrypted sum without

having access to the individual ciphertexts as:

$$\begin{aligned}
&\mathsf{Com}\big(\mathsf{Enc}_{pk}(m_1), r_1\big) \cdot \mathsf{Com}\big(\mathsf{Enc}_{pk}(m_2), r_2\big) \bmod N^2 \\
&= \big(\mathsf{Enc}_{pk}(m_1)^e \cdot g_m^{r_1}\big) \cdot \big(\mathsf{Enc}_{pk}(m_2)^e \cdot g_m^{r_2}\big) \bmod N^2 \\
&= \big(\mathsf{Enc}_{pk}(m_1) \cdot \mathsf{Enc}_{pk}(m_2)\big)^e \cdot \big(g_m^{r_1+r_2}\big) \bmod N^2 \\
&= \mathsf{Com}\big(\mathsf{Enc}_{pk}(m_1) \cdot \mathsf{Enc}_{pk}(m_2) \bmod N^2, r_1 + r_2\big) \\
&= \mathsf{Com}\big(\mathsf{Enc}_{pk}(m_1 + m_2), r_1 + r_2\big).
\end{aligned} \tag{6.3}$$

Therefore, an auditor can compute and open the commitment for the encrypted sum that they received from the curator by using the accumulated randomness. In our implementation, the analyst plays the role of an auditor but we also enable any party to further verify the protocol by inspecting the public ledger. With a successful commitment opening, an auditor is assured that no errors occurred during the aggregation.

Additionally, we extend this core protocol to support computing the weighted arithmetic mean based on both the participants' private inputs and the public weights. From Eq. 2.3, it follows that the curator can multiply encrypted participant inputs with constant weight values w_i, where $i \in [1, P]$. Based on the previous example, for ciphertexts $\mathsf{Enc}_{pk}(m_1)$ and $\mathsf{Enc}_{pk}(m_2)$, the curator computes $\mathsf{Enc}_{pk}(m_1)^{w_1} \cdot \mathsf{Enc}_{pk}(m_2)^{w_2} \bmod N^2 = \mathsf{Enc}_{pk}(w_1 m_1) \cdot \mathsf{Enc}_{pk}(w_2 m_2) \bmod N^2$, and also sums the commitments' randomness. Finally, the analyst incorporates the weights in the commitments received from the participants by applying the following formula: ${w_i}^e \cdot \mathsf{Com}\big(\mathsf{Enc}_{pk}(m_i), r_i\big) = \big(w_i \cdot \mathsf{Enc}_{pk}(m_i)\big)^e g_m^{r_i} = \mathsf{Com}\big(\mathsf{Enc}_{pk}(w_i m_i), r_i\big)$.

6.3.3 Public Verifiability for Aggregator

Masquerade enables the participants to audit not only the curator, but also the analyst and efficiently verify that the published sum is the correct decryption of the ciphertext that the curator published to the ledger. We adopt the ZKP protocol for proving ciphertext equality over N^{s+1} modulus with $s = 1$ from [88] and we apply the Fiat-Shamir heuristic in the random oracle model to convert the ZKP to

a non-interactive variant. Let ct_{sum} be the encrypted result and sum be the decryption published by the analyst. We further provide public verifiability that the analyst decrypted and published the correct result: The analyst first encrypts sum using randomness ρ into ct'_{sum}. Consecutively, the analyst computes a ciphertext ct_0, which is the homomorphic result of ct_{sum} minus ct'_{sum}, and posts ct_0, ct'_{sum}, along with the randomness ρ to the ledger. By construction, if ct_{sum} is an encryption of sum, then ct_0 is an encryption of zero. Finally, the analyst uses the aforementioned non-interactive ZKP and provides public verifiability that ct_0 is an encryption of zero, which means that ct_{sum} is an encryption of sum. Notably, it is not possible for the analyst to cheat and publish a wrong sum as ct_0 would not be an encryption of zero and this would break the soundness of the ZKP [88].

6.3.4 Protecting Against Malicious Clients

As participants contribute their data encrypted, the curator has no way of detecting if a malicious client performs a data pollution attack by sending an encryption of an overwhelmingly large number. Masquerade bounds the influence of each participant in the aggregate results by restricting the values they can input into the protocol. Specifically, to mitigate such attacks, we incorporate two *non-interactive* ZKP protocols, namely a *range proof* and a *set-membership proof*. The former enables the participants to prove that their Paillier ciphertext encrypts a message m that lies within a valid range, i.e., $m \in [0, \ell]$ [52,164], while the latter proves that m lies within a set of messages, i.e., $m \in \{m_1, m_2, \ldots, m_k\}$ [20,88]. In both cases, the client manages to convince the curator by only showing the Paillier encryption of m without leaking any additional information about m.

As discussed in Section 2.2.3, ZKP protocols must satisfy three basic properties: *completeness*, *soundness*, and *zero-knowledge*. Completeness guarantees that an honest participant ($\mathcal{P}$) can always convince the curator ($\mathcal{V}$) about a valid statement. Soundness guarantees that if the curator is honest and the participant's private data are not well-formed (i.e., are not within a pre-specified range or a set of public messages),

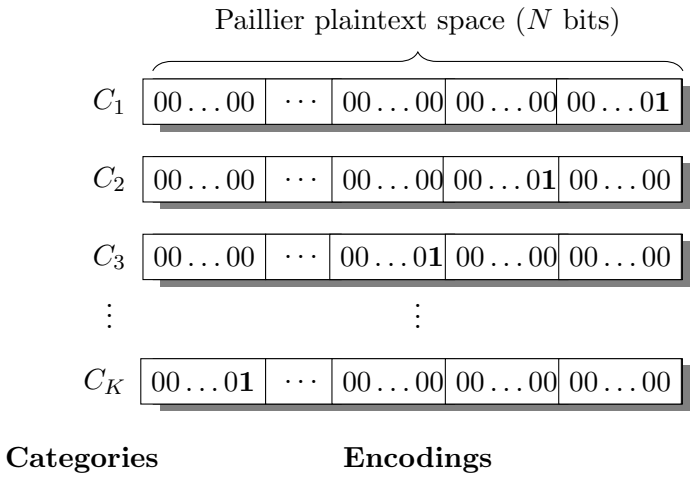
then the curator will reject the proof with overwhelming probability. We formalize the soundness by introducing a security parameter t: the probability of an adversary convincing an honest verifier in range proofs is 2^{-t}, and in set-membership proofs it is A^{-t}, where A is the size of the randomness used for the protocol. Finally, zero-knowledge guarantees that if both the participant and the curator are honest, then the proof does not reveal anything about the private participant's data, except that they are well-formed.

We utilize the zero-knowledge range proof protocol for quantitative aggregations, while the set-membership proof is used to assert that the participants select a message from some predefined categories (described in section 6.3.5). We instantiate these two ZKP protocols as public coins in the random oracle model [28] via the Fiat-Shamir heuristic to eliminate the interaction and transform the interactive proofs to their *non-interactive* variants [106]. Therefore, the data curator verifies the correctness of the proof for the ciphertext sent by each participant and homomorphically increments the sum if and only if the proof was accepted. In Fig. 6.2, the proof for the participant i is depicted as π_i, whereas the prover and verifier algorithms are shown as Prove and Verify, respectively.

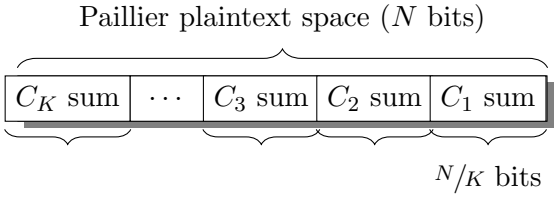
6.3.5 Enabling Categorical Data Aggregation

Masquerade naturally supports the aggregation of quantitative data. On the other hand, categorical variables represent types of data that may be divided into groups, where each group should be encoded and accumulated separately. For instance, an aggregation based on four age groups (e.g., "infant", "child", "adolescent" and "adult") requires four counters, one for each category. In categorical data, each participant's influence is limited since they can contribute at most one to the sum of a category. For example, if the attribute type is "age group", each participant may only contribute to one of the categories and increment that counter (e.g., "adult").

Taking these two requirements into account, we design a specialized encoding to incorporate different categories into one encrypted message. A possible solution



(a) Encodings for K categories.



(b) Accumulated sums for each category.

Figure 6.4: Histograms Overview. We can divide the Paillier plaintext space into different sections, each of which corresponds to a different category. Homomorphic addition of ciphertexts will result into summing the bits from each category and end up with individual accumulators.

for categorical variables would be to use multiple invocations of our quantitative PDA protocol so that every invocation would aggregate one category. Each participant could contribute an encryption of either a "one" or a "zero", depending on their selection, and also prove in zero-knowledge that the submitted ciphertext is an encryption of a binary number. Unfortunately, in cases that the response should be "one-hot encoded" (such as the "age group"), this solution is not sufficient since it would enable malicious participants to contribute to more than one category and tamper with the aggregation result. Although in each invocation they could submit a valid encryption and a proof, the set of their responses could be invalid (e.g., submit multiple encryptions of True).

Masquerade introduces an encoding that enforces the participants to select 1-out-of-K categories using a single message that encodes both the selected category as

well as the remaining $K-1$ non-selected categories. We encode each category as a power of $2^{N/K}$, where N are the bits of the plaintext: specifically, the first category is represented as $(2^{N/K})^0 = 1$, the second category is represented as $(2^{N/K})^1 = 2^{N/K}$, and so on. In Fig. 6.4(a) we illustrate these encodings. We remark that performing homomorphic addition between ciphertexts from the same category will increment the N/K-bit counter for the specific category, and leave the counters for the $K-1$ remaining categories unmodified. The accumulations for each selected attribute create a one-dimensional histogram with K categories, as illustrated in Fig. 6.4(b). We complement our protocol with the non-interactive set-membership proof described in section 6.3.4 to guarantee benevolent client behavior; in this case, each participant encrypts one of K possible messages (i.e., powers of $2^{N/K}$) and creates a zero-knowledge set membership proof that the ciphertext is the encryption of one of these powers.

Using the same number of plaintext bits, our protocol also supports responses that enable the participants to select multiple categories at the same time, if those are valid answers. To support such messages, the set-membership proof includes all the possible combinations of valid responses. For instance, a response for both C_1 and C_3 from Fig. 6.4(a) would be in the form $00\ldots00 \mid \cdots \mid 00\ldots01 \mid 00\ldots00 \mid 00\ldots01$. Adding the above response with another response for C_1 will result in $00\ldots00 \mid \cdots \mid 00\ldots01 \mid 00\ldots00 \mid 00\ldots10$, which accumulates both responses.

Notably, by having large plaintext sizes our encoding can also support multidimensional histograms. For instance, extending the example of age groups, the analyst can also consider if a participant is a smartphone user. We use these two variables (i.e., "age groups" and "smartphone user") to form a two-dimensional array with all combinations and encode each index in the array into a unique category from Fig. 6.4(a). Since "age groups" has four possible values and "smartphone user" has two, the number of categories is $K = 8$. To increase the number of categories and preserve enough bits for each accumulation, a larger plaintext space can be used by increasing N. This introduces a trade-off between the number of categories and the performance of our protocol since larger modulus sizes may increase performance overheads.

6.3.6 Security Sketch

Theorem 2. *Assuming that Eq. 6.1 is a secure commitment scheme with multiplicative homomorphism and satisfies the hiding and binding properties, the Paillier cryptosystem is semantically secure against chosen-plaintext attacks (IND-CPA), and the ZKPs satisfy the three properties from section 2.2.3, then Masquerade is a secure PDA protocol.*

Participant is corrupt. A malicious participant may attempt to tamper with the analysis by encrypting an invalid input. Observe that due to the ZKP soundness an honest curator will always reject malformed proofs, except with negligible probability. Additionally, a malicious participant cannot generate an invalid commitment (i.e., to a different ciphertext) due to the binding property and the curator will always reject invalid commitments. Thus, Masquerade preemptively rejects malicious clients.

Curator is corrupt. It is easy to observe that the participants' data are protected by a passive-corrupt curator since neither Paillier ciphertexts nor the ZKPs reveal anything about the plaintexts. In the case of a fail-corrupt curator that introduces non-adversarial or random faults, our homomorphic commitment provides public verifiability without compromising the participants' privacy and allows a semi-honest analyst and any auditor to verify the integrity of the result.

Analyst is corrupt. Since the analyst only learns the encrypted sum (and thus the final result) and nothing is revealed about each participant's data points. Finally, the analyst proves the correct decryption of ct_{sum}, providing public verifiability.

Auditor is corrupt. Auditors only inspect the protocol and verify its correctness. Malicious auditors cannot affect the protocol as they do not provide any inputs.

6.4 Experimental Evaluations

Our evaluations assess the performance of our protocol and understand how it scales with an increasing number of participants. We expect Masquerade to scale linearly to the number of clients and the zero-knowledge protocols to dominate the

execution time, especially as the soundness parameter t increases. Finally, the set-membership proof is more computationally intensive than the range proof and thus we expect the quantitative studies to be faster than categorical.

6.4.1 Experimental Setup

For our experimental evaluation of the curator and the analyst, we used two t3.2xlarge AWS EC2 instances running with eight virtual processors up to 2.5 GHz and 32 GB RAM. For the experiments evaluating the participants, we used a Lenovo laptop with an i7-8650U CPU running at 1.90 GHz, and we have implemented our Masquerade framework in Go 1.16. In our experiments, the latency between the curator and the analyst was negligible (about 0.014 ms) as both are hosted in AWS instances. The communication between the participants and the AWS instances (curator and analyst) is 9.65 ms on average. The back-end of the curator is fully parallelizable and can take advantage of all the available threads of the host. For every participant, the curator spawns a new thread to verify the zero-knowledge proof of the participant and aggregates their private data only if the ZKP verification was successful. This optimization provides a speed-up proportional (roughly) to the number of parallel cores of the curator. Without loss of generality, in our implementation the analyst maintains a public append-only ledger. The ledger can also be maintained by any trusted third party, or be distributed between the analyst and the curator. Moreover, Masquerade supports ledgers deployed on a public blockchain: our implementation offers end-to-end support for publishing commitments on Ethereum using the Ganache-CLI, and was evaluated on the Ropsten testnet. Finally, both the Paillier cryptosystem and our commitment scheme are instantiated with a 4096-bit modulus N^2 following the key size recommendations of [18].

6.4.2 Performance Evaluation

In Fig. 6.5 we present the *online protocol costs* of Masquerade assuming both honest and malicious participants. In the former – *baseline* – case, the participants'

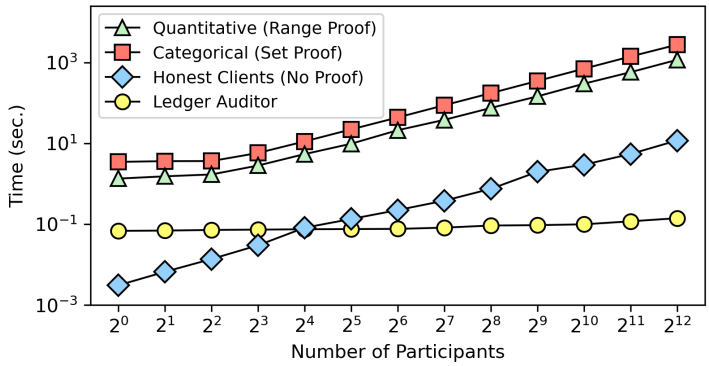


Figure 6.5: **Masquerade Performance.** Time measurements in seconds for an increasing number of participants from 1 to 2^{12}. The timings for malicious participants use $t = 60$ and $K = 4$ and depend on the type of study (quantitative or categorical), while for honest participants the overheads do not depend on the type of study as ZKPs are omitted. Finally, the ledger auditor performance is almost constant as it involves fast modular multiplications.

ciphertexts are well-formed and the curator accepts all the encrypted data directly, eliminating the need for creating and verifying ZKPs. We use the timings of this simplified protocol as a baseline to demonstrate the trade-off of the zero-knowledge protocols for the quantitative and categorical studies (note, the honest participant cost does not depend on the type of study). Fig. 6.5 demonstrates the experimental timings for quantitative variables (i.e., green triangles), categorical variables (i.e., red squares), a baseline for honest participants (i.e., blue diamonds), and the timing for an auditor (i.e., yellow circles) with an increasing number of participants and soundness parameter $t = 60$ bits.

The experimental timings include the computation performed by the analyst and the curator until the result of the aggregation is published by the analyst. We note that Fig. 6.5 does not include the offline cost for key-generation or participant cost (these are discussed later in this Section) since all the participants can pre-compute their ciphertexts, proofs, and commitments before engaging in the protocol. Finally, the timings also include the latency, which has negligible performance cost compared to the ZK-related operations. For the reported cost of the categorical study in Fig. 6.5 we used a set with $K = 4$ classes. As expected, for both types of studies we observe

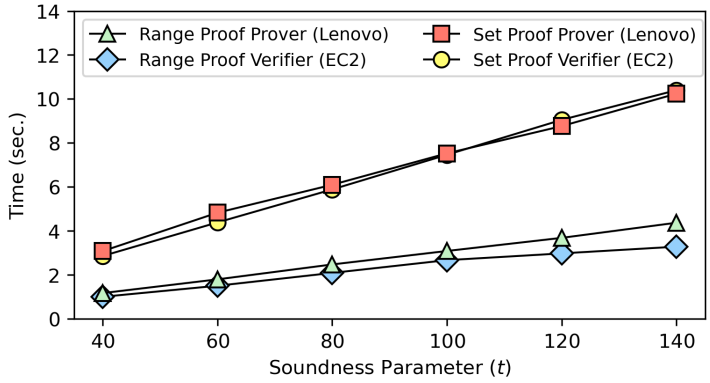


Figure 6.6: **Zero-Knowledge Proofs Performance.** Time measurements in seconds for both range and set-membership proofs for the prover and the verifier with an increasing soundness parameter t in bits and $K = 4$.

a linear increase to the total aggregation cost as the number of participants grows. Additionally, for up to 8 participants the cost of the quantitative and categorical remains somewhat constant, due to the thread parallelization in the curator's back-end. When we compare both trends that use ZKPs with the trend of "Honest Participants", we observe that protecting against malicious participants incurs about two orders of magnitude additional overhead. Likewise, the quantitative study (i.e., green triangles) is about half an order of magnitude faster than the categorical one (i.e., red squares), which is attributed to the complexity of the set-membership [20]. Finally, we observe that the time to audit the ledger is almost constant with an increasing number of participants as it mostly comprises modular multiplications. Next, we analyze the individual performance of the participants, the curator, the analyst, and the auditor and discuss how each one of them affects the total performance of our protocol.

Participant Perspective. The computation cost of each participant includes the Paillier encryption cost, the ZKP cost, and the commitment cost. Our protocol enables the participants to pre-compute everything offline. Both the Paillier encryption cost and the cost of committing to encrypted data are negligible compared to the ZKP generation. Fig. 6.6 shows the proof generation time for participants for both *range* and *set-membership* proofs with varying soundness parameters t. Similarly, Table 6.1 shows the ZKP size in KBytes with an increasing t for range and set-membership

proofs.

Both the computation and space costs of the range proofs scale linearly to the soundness parameter t. Similarly, the computation and space costs of the set-membership proofs scale linearly to t, but are also affected by the number k of messages in the set. In Table 6.2 we show the timings for $\mathcal{P}$ and $\mathcal{V}$ for the set-membership proof with increasing set sizes. Our results show that the performance cost of both $\mathcal{P}$ and $\mathcal{V}$ increases linearly to the set size.

Smartphone participants. We further evaluate the participant costs for soundness $t = 40$ using a smartphone equipped with a Qualcomm Snapdragon 865 at 2.84 GHz. As discussed, this cost is dominated by $\mathcal{P}$: the cost for a range proof is 2.14 seconds while a set-membership proof with $K = 4$ takes 4.78 seconds. In all cases, the commitment cost was negligible at just 78 ms. This analysis shows using a modern smartphone instead of a laptop (c.f. Fig. 6.6 prover costs for $t = 60$) incurs very similar overheads.

Curator Perspective. Assuming P participants, the total curator overhead corresponds to: (1) the verification cost for P proofs, and (2) P homomorphic additions (Eq. 2.2). Like the proof generation shown in Fig. 6.6, the verification cost of both proof types scales linearly to the soundness parameter. As shown in Table 6.2, set-membership verification incurs a linear overhead to the set of valid messages size. Moreover, the verification cost for set-membership is about half an order of magnitude higher compared to range ZKP due to the increased complexity of [20]. This difference justifies why quantitative studies are faster than categorical (as reported in Fig. 6.5).

In either study type, the curator verifies one proof for each of the participants and performs a homomorphic addition if the proof is correct. The overall communication cost of the curator is P times the proof size (Table 6.1), as well as the cost of

Table 6.1: Size of the Zero-knowledge Proof Protocols.

Soundness Parameter (t)	**40**	**60**	**80**	**100**	**120**	**140**
Range Proof (KB)	36.0	57.4	78.1	103.1	118.4	142.2
Set Proof (KB)	309.6	464.5	619.3	774.1	929.0	1083.9

Table 6.2: Set-Membership Proof Timings with an Increasing Number of Set Elements for Soundness $t = 60$.

Set Size (K)	**2**	**4**	**8**	**16**	**32**	**64**
Prover Time (Lenovo) (sec.)	1.92	2.94	7.92	14.61	30.55	63.83
Verifier Time (EC2) (sec.)	2.08	2.61	6.97	13.26	31.93	65.14

transmitting the final results to the analyst (i.e., ct_{sum} and r_{sum}). Likewise, without any optimizations, the curator's performance overhead is P times the cost for one participant; using 8 parallel threads, the curator's performance is about 5.3 times faster. Note that the total protocol performance is dominated by the curator's overhead.

Analyst Perspective. The timing cost of the analyst can be attributed to: (1) the Paillier key generation, (2) the P multiplications and additions of the commitments and the random parameters, respectively, and (3) the constant costs of opening the commitment and decrypting the aggregation result. The key generation offline cost involves generating two large prime numbers and a group element of maximum order. For 2048, 4096, and 8192-bit plaintexts (i.e., N bitsize), key generation takes 43.1 milliseconds, 202 milliseconds, and 1.27 seconds, respectively.

In our implementation of Masquerade, the analyst maintains the public ledger. Thus, the communication cost of the analyst entails receiving the individual commitments from each participant, as well as receive a list of identifiers from the curator corresponding to the participants whose ZKPs were correct. Performance-wise, in addition to decrypting the encrypted sum, the only additional cost of the analyst is to support public verifiability by aggregating all participant commitments and showing that it correctly opens with the homomorphic sum (Alg. 4 in Fig. 6.3).

Auditor Perspective. Finally, any auditor can access a copy of the ledger in order to verify that the protocol was executed correctly, which allows public verifiability. The auditor uses the accumulated randomness and verifies that the participants' homomorphic commitments open with the encrypted sum. This verification is similar to the one that the analyst performs in Fig. 6.3 Alg. (4), line 2. Lastly, as mentioned in section 6.3.2, the auditor verifies that the published decrypted sum is indeed a correct

decryption of the ciphertext that the curator published to the ledger. We measured the auditor timings in Fig. 6.5 (i.e., yellow circles) with modern Lenovo laptop (the same machine was also used for the participants) and we observe almost constant overhead.

6.5 Related Work

In this section, we discuss several recent works in PDA that rely on cryptographic techniques such as HE, secure multiparty computation (MPC), and differential privacy. Existing solutions introduce different trade-offs between computation and communication costs, depending on the underlying privacy technique they employ. Interestingly, some protocols provide robustness against dropouts, while only a few are secure against malicious clients that provide malformed inputs. Masquerade and only two other works allow the participants to verify the outcome of the protocol. Next, we discuss these existing works and how they compare with Masquerade.

In MPC, multiple parties want to jointly compute a function over their private data while keeping those inputs private [120, 246]. Most common constructions are either based on Yao's garbled circuits or on HE and secret sharing [6, 63, 117, 148, 166]. The latter is more computationally efficient than the former but incurs high communication costs. Such generic MPC protocols can be used for PDA but since they are not optimized solely for this task, they are not as efficient as customized PDA solutions such as Masquerade. Likewise, generic schemes allow the participants to provide *any input* to the aggregation, whereas Masquerade protects against malicious inputs using the two non-interactive ZKPs discussed in section 6.3.4. Both MPC and the works in [90, 161] that use thresholding-based cryptography (such as Shamir's secret-sharing [211]) remain susceptible to participant dropouts. Conversely, Masquerade is immune by design to participants intentionally leaving the computation and offers public verifiability, where third parties can audit the result.

Shi *et al.* [214] proposed a PDA methodology based on random shares and distributed noise generation. The authors of [71] and [142] extended Shi's protocol by handling client dropouts and enabling larger plaintext spaces, respectively. Likewise,

the work in [90] is also based on secret-sharing techniques for privacy-preserving smart meter readings, yet it cannot tolerate participant dropouts. Leontiadis *et al.* [160] extended [142] and eliminated some of the trust assumptions that the latter requires. PUDA [161] is a pairing-based scheme that uses a *trusted dealer* to set up secret keys and enables public verification using homomorphic tags. While these aforementioned protocols support only quantitative variables and any client can corrupt the results, Masquerade offers robust protection against malicious clients, as well as support for both categorical and quantitative studies. The work in [47] features an efficient t-out-of-n secret sharing protocol, however, like most related works, it cannot tolerate malformed inputs from malicious participants. Trinocchio [209] outsources a computation *from a single client* to multiple workers in a privacy-preserving way and generates verifiable guarantees of the correct computation. Trinocchio additionally describes a multi-client version but this variant requires a special key generation and also is susceptible to malicious inputs: if any client provides incorrect information, the computation is aborted. Moreover, among the aforementioned works, only PUDA and Masquerade offer public verifiability.

Prio [83] is based on multiple non-colluding servers and is the most closely related to our scheme. It introduces a technique called secret-shared non-interactive proofs (SNIPs) to validate that clients' inputs are the encryption of either a "0" or a "1". To aggregate longer values than just one bit, Prio encodes the participants' private data as a sequence of bits, where each bit is verified by a different SNIP, and computes the sum of these encodings. Therefore, since SNIPs inherently prove one bit, to prove messages of bigger sizes the proof size increases linearly to the number of bits. Notably, the range proofs in Masquerade have *constant size and time*, whereas our set-membership proofs *scale linearly* to the set elements. For very small client inputs, the design of Prio enables fast participant timings: For instance, the authors report that for one-bit up to four-bit integers, the participant timing varies from 0.01 to 1 second, depending on the number of the multiplication gates on the SNIP circuit. We remark that Masquerade offers better scalability, as our plaintext dynamic range is in the order

Table 6.3: A comparison with existing PDA schemes based on the cryptographic technique they utilize, the type of variables they support and their robustness against dropping participants and malicious inputs.

Approach	Cryptographic Technique	Quant. Vars	Categ. Vars	Robust Against Dropouts	Public Verif.	Robust Against Malicious Inputs[§]	Proof Scaling Quantitative/ Categorical
Shi *et al.* [214]	THE[*] & Diff. Privacy	●	○	○	○	○	N/A
Chan *et al.* [71]	THE & Diff. Privacy	●	○	●	○	○	N/A
Joye *et al.* [142]	THE	●	○	○	○	○	N/A
Danezis *et al.* [90]	THE	●	○	○	○	○	N/A
Leontiadis *et al.* [160]	THE	●	○	●	○	○	N/A
Leontiadis *et al.* (PUDA) [161]	THE	●	○	○	●	○	N/A
Bonawitz *et al.* [47]	Pairwise Masking	●	○	●	○	○	N/A
Burkhart *et al.* (SEPIA) [63]	Secret-Sh., Generic MPC	●	○	●	○	●	Quadratic / N/A
Giannopoulos *et al.* [117]	Secret-Sh., Generic MPC	●	●	●	○	○	N/A
Narula *et al.* (zkLedger)[†] [190]	Pedersen Commitments	●	○	●	●	●	Quadratic / N/A
Corrigan-Gibbs *et al.* (Prio)[‡] [83]	Secret-Sh.	●	●	●	○	●	Linear / Linear
This Work (Masquerade)	PHE & Commitments	●	●	●	●	●	Constant / Linear

[†] zkLedger is the only related work that achieves both public verifiability and being robust against malicious inputs, however, the transaction verification is quadratic to the number of participants causing major scalability issues (the authors only experiment with at most 20 participants). Additionally, zkLedger does not support categorical variables.

[‡] Prio introduces a mechanism called affine-aggregatable encodings ("AFEs") to encode a series of bits as quantitative or categorical variables. Prio does not guarantee public verifiability and each client only communicates with the servers (no support for a distributed ledger).

[*] Threshold Homomorphic Encryption (THE): The private key is shared between n participating parties and in order to decrypt a message at least t-out-of-n parties are required.

[§] Our protocol utilizes non-interactive zero-knowledge proof protocols to protect against malicious inputs.

of thousands of bits (e.g., 2048 bits in our experiments), which can encode significantly more information. While the authors of Prio do not report any timings for categorical studies (which can be theoretically supported with modified proofs), their approach introduces additional overhead since each participant must prove that the sum of all the bits is exactly one. Finally, Prio does not operate on a public ledger and does not offer any public verifiability protections, contrary to our Masquerade protocol.

The work in [190] introduced zkLedger, a method that protects participants' privacy and provides public verifiability. The participants in zkLedger rely on Pedersen commitments [197] and Schnorr-type ZKPs [208] to publish their transactions on a public ledger. Then an auditor can combine all the public commitments and verify their correctness. Likewise, in Masquerade we allow the analyst to audit the commitments, but since everything is public, any participant or third party can also audit the committed result. Nevertheless, zkLedger incurs significant overheads since

transaction verification costs are *quadratic to the number of participants*, so practicality is significantly impacted in studies with more than 10-20 participants. Conversely, our experiments show that Masquerade scales gracefully for thousands of participants.

Table 6.3 summarizes all notable related works and compares with ours. Interestingly, these approaches use a broad range of cryptographic techniques, yet all of them have homomorphic properties. We observe that most of these constructions are tailored to quantitative studies, whereas only our protocol and Prio [83] provide encodings to represent categorical variables (e.g., Likert scales). Moreover, only PUDA, zkLedger, and Masquerade allow practical auditing and public verifiability, which is an important property for real-world applications. Although several works have focused on malicious participants intentionally dropping out of the protocol, only our protocol, SEPIA, Prio, and zkLedger protect against malicious inputs using non-interactive ZKPs; however, only Masquerade offers constant ZKP costs for quantitative studies.

The security and threat models of the aforementioned existing works vary based on the number of computing parties, the types of studies they support, and their robustness against malicious participants. The protocols in [71, 142, 214] use a single untrusted aggregator, and assume the *aggregator obliviousness* model which ensures that the aggregator learns only the sum of users' inputs and nothing else. The authors of [161] extended the aggregator obliviousness model and introduce the concept of *aggregate unforgeability*, which assumes that one party performs the aggregation and another party decrypts and publishes the final result. Aggregate unforgeability ensures that the aggregator cannot convince the data analyzer to accept a bogus sum. Similarly to our work, in this model the two parties should never collude. In [47], the authors introduce three different security models, one which ensures security only against the clients, one which ensures security only against the server, and a mixed security model for a threshold of colluding clients with the server. The work in [63] uses the honest-but-curious model and is secure as long as no more than half of the computing nodes collude, while the authors of [117] use the Sharemind MPC framework with three honest-but-curious non-colluding nodes. Lastly, the protocol in [190] is secure against

malicious input providers (e.g., banks) and keeps the transactions private as long as that the sender, the receiver, and the auditor do not collude, whereas the authors of [83] use a small set of servers (e.g., three) and protects the client privacy if not all the servers collude. Like Masquerade, most related works rely on two or more non-colluding servers. However, as shown in Table 6.3, only Masquerade offers public verifiability while protecting against malicious inputs and scales with an increasing number of participants.

6.6 Concluding Remarks

We presented Masquerade, a new scheme for private data aggregation that ensures that participants' inputs are kept private and the analyst only learns the final aggregate result. We extended our protocol to work both with quantitative and categorical variables, enabling a variety of different studies. Our two non-interactive ZKP protocols complement our scheme to defend against malicious participants by verifying that the encrypted data are well-formed and that each participant has limited influence on the protocol. Masquerade publishes our novel *multiplicative homomorphic* commitments on a ledger to enable public verifiability, which is an essential property for real-world applications where participants need to assess the correctness of the encrypted sum and verify that their response is included in the final result. Moreover, our protocol's public verifiability allows the analyst to prove that the announced decryption of the encrypted aggregate is correct, which allows any participant to audit the final results. Our experiments demonstrate that Masquerade scales gracefully to thousands of participants, and further supports smartphone clients.

Chapter 7

PLASMA: PRIVATE, LIGHTWEIGHT AGGREGATED STATISTICS AGAINST MALICIOUS ADVERSARIES

7.1 Introduction

In today's technology-driven world, companies are constantly collecting user data to perform analysis, compute statistics, expose patterns in user behaviors, and apply them to improve their products [68,103,134,162,198]. Common analysis practices resort to histograms, where client data are aggregated together in predefined and non-overlapping buckets. Each bucket may represent a quantitative range (e.g., salary) or a categorical value (e.g., profession). The resulting histogram displays the frequencies of each bucket based on multiple aggregated participant responses.

Private Histograms. When computing histograms, it is crucial to maintain client privacy, such as preventing data collection servers from inferring additional information about the clients. Existing solutions for privacy-preserving histograms solve this problem efficiently [27, 49, 83], given a relatively small number of buckets. Nevertheless, histograms are resource-intensive on the server side when the goal is to find popular entries among the clients' inputs. For instance, assume clients that hold GPS coordinates of their location and servers aiming to discover crowded areas without compromising client privacy. The naive solution of creating a histogram over all possible inputs results in sparsely populated sets, which wastes server-side computational power due to sparse inputs. Conversely, in an optimal solution, the server computation should scale with the most popular inputs, instead of all possible ones.

Private Heavy-Hitters. This problem is addressed by the concept of "heavy hitters". $\mathcal{T}$-heavy hitters allow computing the $\mathcal{T}$ most popular responses (for a given threshold

$\mathcal{T}$) among clients' inputs and have a broad range of applications: from finding popular websites that users visit or malicious URLs that cause browsers to crash [49, 132], to discovering commonly used passwords [188], learning new words typed by users and identifying frequently used emojis [105], to name a few. Private heavy-hitters allow computing these results while also preserving client privacy. Existing protocols (such as [8, 46, 49, 188]) only focus on the "popular" inputs and disregard other inputs that appear less than $\mathcal{T}$ times (i.e., they are pruned by the protocol). This renders private heavy hitters a suitable candidate for finding the most common client entries, such as computing crowded areas using client-provided GPS coordinates.

Table 7.1: Threat model comparisons, client input validation, and server-to-server communication.

Protocol	Correctness & Privacy Against Malicious Corruption			Client Input Validation	Low Server-to-Server Communication	No. of Servers
	Clients	Server	Server & Clients			
DPF [55, 56, 118]	●	◐†	○	○	○	2+
Poplar (IDPF) [49]	●	◐†	○	●	○	2
Bucketization (DP) [8]	●	◐†	○	●	○	2-3
MPC-based [46]	◐‡	◐†	○	○	○	3
Sorting-based [13, 138]	●	●	●	●	○	3
PLASMA (this work)	●	●	●	●	●	3

† These works only preserve privacy against a malicious server but not correctness.
‡ [46] is susceptible to data poisoning attacks by malicious clients or malicious servers. Privacy of honest clients is preserved.

Different Approaches. The literature considers the setting where two or more servers collect client inputs and run the private heavy-hitters protocol. A notable approach based on differential privacy (DP) is [8] (we discuss DP-based solutions in Section 7.1.2). While these protocols are computationally fast, they are limited to DP-based privacy guarantees for the client. Likewise, MPC-based solutions [46] employ general-purpose secure computation frameworks (e.g., MP-SPDZ [148], SCALE-MAMBA [5], Sharemind [45]), so these methods fall short in terms of practicality. Thus, recent works introduced custom MPC-based techniques for private heavy-hitters [13, 139]. The underlying protocols perform secure sorting of client inputs under MPC [13, 139] and then

aggregate the sorted data, guaranteeing that private inputs remain hidden when a majority of the servers are honest. However, the communication of all aforementioned solutions is linearly dependent on the number of clients, resulting in high server-to-server communication costs.

Distributed point functions (DPFs) [55] offer an alternative approach for private histograms. Informally, DPFs allow a client to send succinct shares of a point function corresponding to their private inputs to two or more servers. The servers then use these shares to locally evaluate the function over the entire input space and add the resulting outputs to obtain additive shares of a histogram.

Poplar [49] builds upon the DPF approach by introducing incremental DPFs (IDPF). It provides an IDPF-based solution for private heavy-hitters in the two-server setting, and their server-to-server communication depends on the input string length in semi-honest security. For security against malicious clients, the servers validate every client's input so that malformed inputs are preemptively discarded from the computation. This is referred to as *client input validation* and it prevents malicious clients from causing an abort in the protocol. To do so, Poplar requires additional checks, which cause the server-to-server communication to scale linearly with the total number of clients. As a result, their concrete server-to-server communication is large. Sabre [232] uses multi-verifier MPC-in-the-head that attests to the well-formedness of DPFs but does not focus on heavy hitters.

Motivation. Since all aforementioned solutions incur server-to-server communication that scales linearly with the number of clients, where the concrete communication cost is large, they are prohibitive for most real-world applications that require millions of clients for data collection. Hence, it is desirable that the concrete server-to-server communication is low, even for a large number of clients. Likewise, neither Poplar nor the DP-based solutions [8] tolerate additive attacks from a malicious server, which results in incorrect outputs when one of the servers does not follow the protocol steps. More formally, they fail to provide both correctness and privacy against the collusion of a malicious server and malicious clients. In this regard, we ask the following motivating

question:

Can we obtain a private heavy-hitters protocol with low concrete server-to-server communication that is secure against malicious clients and a malicious server?

7.1.1 Our Contributions

We answer the aforementioned question by proposing PLASMA, a framework for private and lightweight statistics that provides security against a malicious server and malicious clients. Our main contributions are summarized as follows:

Verifiable incremental DPF (VIDPF). First, we introduce a new primitive called VIDPF, which builds upon incremental DPFs (IDPF) [49] and verifiable DPFs (VDPF) [94]. VIDPF allows us to verify that clients' inputs are valid by relying on hashing while preserving the client's input privacy. We also propose a novel way to verify that IDPF keys are *"one-hot"* - i.e., they have a single non-zero evaluation path (containing the same value along the path) by solely relying on hashing. This is of independent interest and can be used to improve earlier results in [49, 92, 93]. Previous protocols solved this problem using Fully-Linear Proofs (FLPs) [48, 93] or expensive sketching that involves information-theoretic MACs [49, 50, 92]. More specifically, [93] uses FLPs in each level to verify that the client's input is one-hot, resulting in significant communication overhead as each FLP entails a large proof. Conversely, our checks for one-hot vectors do not require field multiplications, only additions and hashes which allow us to batch-verify multiple inputs together.

Batched Consistency Check. Next, we introduce a novel batched consistency check that allows us to drastically reduce server-to-server communication. At a high level, we validate the inputs of ℓ clients using a Merkle tree and identify the malformed ones using logarithmic (in the total number of clients denoted as ℓ) communication. This optimization reduces the dependency of our server-to-server communication on the total number of clients from $\mathcal{O}(\ell)$ to $\mathcal{O}(\ell'(\log_2 \frac{\ell}{\ell'}))$ number of hashes where there are ℓ' malicious clients, yielding a concrete improvement over the state-of-the-art (as

reported in our experiments), even in the presence of malicious clients. Here, ℓ' is the number of corrupt clients who provide malformed inputs during the protocol execution and it does not need to be a priori bounded. In case $\ell' = 0$, then our servers only exchange a pair of hashes. Our communication cost remains low even when a constant fraction (e.g., 10%) of the clients are malicious.

PLASMA framework. We combine these new primitives to construct PLASMA, a protocol for private histograms and heavy hitters in the three-server setting that guarantees security against a malicious server and malicious clients while maintaining low server-to-server communication. PLASMA relies only on efficient hashing and cheap field additions rather than expensive general-purpose MPC or field multiplications. Due to our novel VIDPF primitive, PLASMA outperforms Poplar with regard to runtime by a factor of $5 - 10\times$ over WAN for $\mathcal{T} = 1\%$ of the clients. In the same setting, our batched consistency check optimization enables us to drastically outperform both Poplar and the sorting-based protocol of [13] in terms of server-to-server communication by a factor of $35\times$ and $45\times$, respectively. For these conditions, we further analyzed the monetary cost of PLASMA, [13], and Poplar and report that PLASMA is more than $2.5\times$ and $4\times$ cheaper than these works, respectively.

Applications. We evaluate PLASMA for two applications: a) detecting frequently visited URLs, and b) identifying popular coordinates.

Popular URLs. A prominent application (discussed both in [13] and [49]) is identifying which URLs crash the clients' browsers more frequently. Each client has a string of n bits that represents the last URL that crashed their browser. In our evaluations (Section 7.6), we consider $n = 256$ bits, which is sufficient for standard domain names, and compute the heavy hitter URLs that caused more than 1% of client browsers to crash. We perform the task over WAN in approximately 5 minutes for 10^6 clients, while incurring less than 1 GB of server-to-server communication (less than $1 in total cost).

Popular GPS coordinates. We demonstrate a new application where PLASMA identifies popular geographic locations without sacrificing user privacy. This can be beneficial with traffic avoidance, restaurant recommendations, as well as advertising (e.g., businesses may identify crowded shopping areas and target their marketing efforts), while ensuring the GPS coordinates of the users remain private to the servers. Likewise, ride-sharing services can enhance vehicle distribution in busy areas and proactively dispatch more drivers during rush hour. This is possible by encoding GPS coordinates as 64-bit strings using *plus codes* [167]. We compute the heavy hitter plus codes for a threshold $\mathcal{T} = 1\%$ in under 2 minutes over WAN across 10^6 clients, while incurring very minimal server-to-server communication with \$0.3 in total monetary costs.

Extensions. We also discuss how to extend PLASMA to obtain fairness against a malicious adversary that corrupts one server and an arbitrary number of clients. PLASMA is the first work to consider different thresholds for heavy hitters based on pre-agreed prefixes by the servers, allowing for more elaborate private statistics, such as the GPS application, where different coordinates (e.g. highways and suburban roads) have different congestion thresholds.

7.1.2 Related Work

We now discuss relevant works for private heavy hitters. They can be classified into four main groups: those based on DPFs, those based on differential privacy (DP), those based on MPC sorting, and finally those based on general-purpose MPC. A comparison of our protocol with related works can be found in Table 7.1.

DPF-based. Distributed point functions [55] offer a straightforward solution for private histograms but they fail for heavy hitters due to the blowup in key size, as the client would need to send new keys for each level. This was addressed by Poplar [49], which uses two non-colluding servers and introduces the notion of IDPFs to allow efficient evaluation of strings based on prefixes. Poplar's threat model is robust against malicious clients but remains susceptible to additive attacks by a malicious server.

Therefore, as the servers reconstruct the output, a malicious server can add arbitrary noise to the result without the honest server realizing it. The recent work of [93] proposes a framework for secure data aggregation and they improve the clients' consistency checks in Poplar and Prio [83]. However, their threat model does not address additive attacks from a malicious server either. In contrast, PLASMA provides security against both a malicious server and malicious clients by adding one additional server. Also, Poplar still leaks some information about the heavy hitter prefixes to the servers as they reconstruct the roots of the paths before they prune them. On the other hand, PLASMA performs a secure comparison over the secret shares and either keeps the node with its subtree if $\mathcal{T} > count$, or prunes the subtree.

Differential Privacy-based. There is also a body of work based on local DP and randomized responses for heavy hitters [19, 200, 253]. These techniques use a single server to collect data from clients. Therefore, this method introduces a trade-off between utility and privacy, as it leaks some information about clients' private data to the server. In contrast, other methods that provide stronger privacy guarantees require at least two not-colluding servers. Notably, secure computation-based solutions can be modified to achieve DP either by using local DP or by adding a smaller amount of noise in MPC and achieving higher data utility while maintaining privacy.

Likewise, bucketization [8] computes approximate statistics on a permuted version of the clients' data combined with dummy data that are sampled as differentially private noise. Bucketization ensures security against malicious clients, and similarly to Poplar, it can only guarantee privacy *without correctness* in the presence of a malicious server. In contrast, PLASMA focuses on exact statistics and provides both correctness and privacy against both malicious clients and one malicious server.

Sorting-based. Recent works that rely on secure sorting algorithms construct private heavy-hitter protocols [13, 139] or private ad attribution measurement [68] based on the sorted data. They provide security against malicious servers and clients in the three-server setting, where one of the servers can be malicious. These protocols are

computationally fast over LAN. However, they perform secure sorting under MPC, and as a result, they incur heavy communication overheads and their performance degrades significantly over realistic WAN networks. Notably, PLASMA achieves a 45× improvement in server-to-server communication compared to [13] as shown in Fig. 7.19 for $\mathcal{T} = 1\%$. Moreover, our PLASMA protocol allows different thresholds for heavy hitters based on pre-agreed prefixes (allowing for more elaborate statistics), this is not possible for sorting-based heavy-hitter protocols.

General MPC-based. One could use generic MPC in the honest majority [74, 109] or dishonest majority setting [148] to compute heavy hitters, but *an efficient representation* of the heavy-hitters problem in terms of addition and multiplication gates is not known. In fact, the work by Böhler and Kerschbaum [46] provides a generic MPC-based protocol for computing differentially private heavy hitters. They use MPC frameworks like MP-SPDZ [148] and SCALE-MAMBA [5] to achieve semi-honest and malicious security, but their solution suffers from high communication and slow runtime.

3-Party Computation (3PC) based. Multiple customized 3PC protocols [13, 139] aim to solve the problem of heavy-hitters. These works consider a third server to exploit the faster computation guarantees in the honest majority. Using a third server is realistic setup and it is widely considered both in the industry and academia as it ensures practical deployments with malicious security. Notable examples include the Interoperable Private Attribution (IPA) proposal by Meta and Mozilla [68], JP Morgan's PrimeMatch [198], NTT's heavy-hitters protocol [13], protocols for private advertisement measurement [180], Duoram [231], and Sabre [232], among others. The servers are meant to run across different organizations; for example, they can be hosted by companies and non-profit organizations as mentioned in Section 5 of Google-Apple's Covid Exposure system [9]. Table 7.1 compares our work with state-of-the-art results.

Multiple Sessions. Combining these to achieve both performance and malicious security is an exciting future direction.

7.2 Technical Overview

Threat Model. Our threat model assumes three non-colluding servers $(\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2)$ that run the histogram/heavy-hitters protocol, as well as ℓ clients. The clients provide inputs to the servers and the servers do not have any private input. We assume that an adversary $\mathcal{A}$ maliciously corrupts one of the servers and $\ell' < \ell$ clients.

Clients. Malicious clients may try to deviate from the protocol in order to disproportionally influence the result or even completely corrupt the output of the protocol. PLASMA is robust against malicious clients and PLASMA servers preemptively reject any malformed client input before incorporating it into the computation. The privacy of honest clients is always preserved.

Servers. Similarly, a malicious server may try to deviate from the protocol steps and attempt to learn private user inputs; PLASMA always protects input *privacy* against one malicious server. Another possible attack for a malicious server would be to over-influence or corrupt the result of the protocol. The semi-honest model does not protect correctness against malicious behavior by a server, which is problematic in real-world applications, like advertisement measurements [68] between two companies, where one company may benefit from reporting inflated measurements by introducing undetectable errors. Malicious security ensures that such malicious behaviors are caught. Therefore, parties are forced to behave honestly, hence fostering a transparent environment for computation. Poplar has this limitation while PLASMA protects *correctness* against a malicious server. Hence, PLASMA is *robust* against a malicious server, since it protects both correctness and privacy.

Notation. We denote the computational and statistical security parameters by κ and μ, respectively. Let $\mathsf{PRG} : \{0,1\}^\kappa \to \{0,1\}^{2(\kappa+1)}$ be a pseudorandom generator and $\mathsf{Convert} : \{0,1\}^\kappa \to \mathbb{G}$ be a map converting a random κ-bit string to a pseudorandom group element of $\mathbb{G}$, where $\mathbb{G}$ is an additively homomorphic group of size $|\mathbb{G}| > \ell$. We use $\coloneqq$ for assignment, $\leftarrow \mathcal{D}$ for sampling from distribution $\mathcal{D}$, $=$ for checking equality, and $\|$ for concatenation. For histograms, wedefine a public set $\mathbf{X}$ with m n-bit strings

as $\mathbf{X} \coloneqq \{x_1, x_2, \ldots, x_m\}$ where the ith string is denoted as x_i for $i \in [m]$ and the jth bit in $x_i \in \{0,1\}^n$ is denoted as $x_{i,j}$ for $j \in [n]$. We denote the first L bits of x_i as $x_{i,\leq L} \coloneqq (x_{i,1}, x_{i,2}, \ldots x_{i,L})$ for $L \leq n$. Let $\mathcal{S}_b$ denote the bth server, for $b \in \{0,1,2\}$; we consider $b+1 \coloneqq (b+1) \mod 3$ and $b+2 \coloneqq (b+2) \mod 3$. We assume ℓ clients, each denoted as $\mathcal{C}_i$ for $i \in [\ell]$. For an n-bit string a we represent its bit decomposition as $a_1, \ldots, a_n \in \{0,1\}$. In histograms, each client $\mathcal{C}_i$ has an n-bit input string $\alpha_i \in \mathbf{X}$, for $i \in [\ell]$, while $\alpha_i \in \{0,1\}^n$ in the case of heavy-hitters. We use $\alpha_{i,1}, \ldots \alpha_{i,n} \in \{0,1\}$ to denote the bit representation of the client's input α_i.

7.2.1 Histogram Protocol of Poplar

Poplar first considers the problem of computing private subset histograms. Each client holds an n-bit string α and the servers $\mathcal{S}_0$ and $\mathcal{S}_1$ have a small set $\mathbf{X} \coloneqq \{x_1, x_2, \ldots, x_m\}$ of m n-bit strings. Each client secret shares their input $\alpha \in \mathbf{X}$ using a DPF as $(\mathsf{key}_0, \mathsf{key}_1) \coloneqq \mathrm{DPF.}\mathsf{Gen}(1^\kappa, \alpha, 1, \mathbb{G})$. The client sends key_0 to $\mathcal{S}_0$ and key_1 to $\mathcal{S}_1$. Upon receiving the client key, each server $\mathcal{S}_b$ evaluates the DPF on all m strings of $\mathbf{X}$ as $y_b \coloneqq \{\mathrm{DPF.}\mathsf{Eval}(b, \mathsf{key}_b, x_i)\}_{x_i \in \mathbf{X}}$ and computes a vector of output shares $y_b \in \mathbb{F}^m$, where $\mathbb{F}^m$ represents m field elements. The servers repeat this for multiple clients and aggregate the y_b vectors in a counter vector Y_b. Finally, the servers exchange Y_0 and Y_1 to compute the output histogram as $Y \coloneqq Y_0 + Y_1$. This protocol requires the client to communicate one key to each server and the server-to-server communication is independent of the number of clients since Y_0 and Y_1 are aggregated values. This protocol preserves client privacy.

However, a malicious client can double vote by generating the DPF keys maliciously such that it contains more than one non-zero point or the DPF output at α is greater than 1. To tackle this, Poplar introduces a malicious sketching protocol to ensure that the client inputs are well-formed. It also preserves the client's privacy against a malicious server. However, Poplar allows a malicious server to add an error to its shares of the output without the honest server realizing it. For instance, say $\mathcal{S}_0$ is malicious and introduces additive errors (e.g., $\delta \in \mathbb{F}^m$) in $Y_0' \coloneqq Y_0 + \delta$. That way,

the output Y of the histogram would be biased by δ as $Y \coloneqq Y_0' + Y_1 = Y_0 + Y_1 + \delta$. The honest server $\mathcal{S}_1$ cannot detect such an additive attack, leading to an error in the correctness of the protocol. Moreover, Poplar's server-to-server communication scales linearly with $\mathcal{O}(\ell)$ due to the malicious sketching protocol.

7.2.2 Our Basic Histogram Protocol

We address Poplar's limitations by (1) introducing one additional server, (2) building upon the primitive of verifiable DPF [94], and (3) introducing novel consistency checks in the three-party setting. We claim the following benefits over Poplar:

(a) Robustness against collusion of a malicious server and malicious clients,

(b) Lightweight consistency checks for malicious behavior (using only symmetric operations and field additions),

(c) Server-to-server communication depends logarithmically on the total number of clients.

Our work provides the first maliciously secure protocol whose server-to-server communication is logarithmic in the total number of clients ℓ. Our servers communicate $\mathcal{O}(\ell'(\log_2 \frac{\ell}{\ell'}))$ hashes for consistency checks, where ℓ' is the number of corrupt clients. Similar to Poplar, we ensure input validation against malicious clients (i.e., honest servers preemptively detect inconsistent inputs and discard them). We present the ideas of our histogram protocol, which are crucial for our heavy-hitters protocol in Section 7.2.4.2.

Robustness Against a Malicious Server. The histogram protocol of Poplar is not robust against a malicious server. Hence, we consider a third server $\mathcal{S}_2$ to allow an honest majority to obtain security against one malicious server with improved efficiency. Each client runs three DPF sessions, one between each pair of servers, with independent randomness, but the same input α (i.e., the pairwise evaluation of the DPF keys on point α outputs secret shares of one).

However, adding a third server significantly complicates things as we need to ensure consistency between the three sessions. For instance, we need to check that a malicious client submitted the same input α to all three sessions without revealing it. The client sends the DPF keys for the sessions to the servers and each server obtains two keys. Upon obtaining the DPF keys, each server evaluates the DPF on all input points in $\mathbf{X}$. It is ensured that if the client behaved honestly then at least one of the three sessions will be evaluated honestly since two of the servers are honest. After aggregating all the clients' inputs, the output histogram is reconstructed across the three sessions. If the output is the same between each pair of servers then the servers behaved honestly and that is considered as the output. If the output is inconsistent across a pair of servers then one of the servers behaves maliciously (by launching an additive attack) and the honest servers abort, which provides robustness against the malicious server.

Reducing Server-to-Server Latency. We empirically observed that the server-to-server latency increases if there is pairwise communication between the three servers for consistency checks. There are three server-to-server sessions for each client, and the third server $\mathcal{S}_2$ is involved in two of the three sessions: specifically, sessions $\mathcal{S}_1 - \mathcal{S}_2$ and $\mathcal{S}_2 - \mathcal{S}_0$. The client generates $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(1,0)})$ for session $\mathcal{S}_0 - \mathcal{S}_1$, $(\mathsf{key}_{(1,2)}, \mathsf{key}_{(2,1)})$ for session $\mathcal{S}_1 - \mathcal{S}_2$, and $(\mathsf{key}_{(0,2)}, \mathsf{key}_{(2,0)})$ for session $\mathcal{S}_2 - \mathcal{S}_0$. $\mathcal{S}_0$ receives $\mathsf{key}_{(0,1)}$ and $\mathsf{key}_{(0,2)}$ from the client for sessions $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_2 - \mathcal{S}_0$, respectively. $\mathcal{S}_1$ receives $\mathsf{key}_{(1,0)}$ for session $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathsf{key}_{(1,2)}$ for $\mathcal{S}_1 - \mathcal{S}_2$, while $\mathcal{S}_2$ receives $\mathsf{key}_{(2,1)}$ and $\mathsf{key}_{(2,0)}$ for sessions $\mathcal{S}_1 - \mathcal{S}_2$ and $\mathcal{S}_2 - \mathcal{S}_0$, respectively.

In our optimization, instead of running two sessions in each server, we run all three sessions between $\mathcal{S}_0$ and $\mathcal{S}_1$ and use $\mathcal{S}_2$ as the attestation server. By doing that, we significantly reduce the latency due to the synchronization overhead of the three servers. To enable that, our protocol instructs the client to send $\mathsf{key}_{(2,1)}$ to server $\mathcal{S}_0$ and $\mathsf{key}_{(2,0)}$ to server $\mathcal{S}_1$ respectively. The key distribution process by the client is illustrated in Fig. 7.1.

Our optimization allows $\mathcal{S}_0$ to replicate the computation of $\mathcal{S}_2$ in session $\mathcal{S}_1 - \mathcal{S}_2$

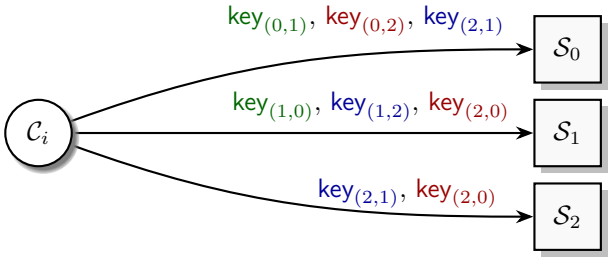


Figure 7.1: Distribution of session keys by client $\mathcal{C}_i$.

(because they both have $\mathsf{key}_{(2,1)}$) and $\mathcal{S}_2$ acts as an attestator by just sending hashes to $\mathcal{S}_1$ for the same messages that $\mathcal{S}_0$ should send. These hashes prevent $\mathcal{S}_0$ from acting maliciously. Similar protocol steps are run by $\mathcal{S}_2$ to attest the $\mathcal{S}_2 - \mathcal{S}_0$ session and prevent $\mathcal{S}_1$ (who is replicating $\mathcal{S}_2$) from acting maliciously. This optimization, shown in Fig. 7.2, allows us to batch-verify all three sessions as a single session between $\mathcal{S}_0$ and $\mathcal{S}_1$ using hashes.

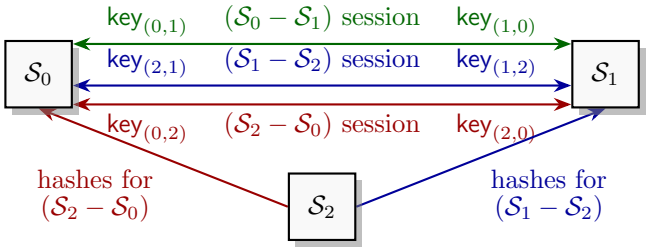


Figure 7.2: Session keys and attestation by $\mathcal{S}_2$.

Client Input Validation. The above protocol assumes that the client computes the DPF evaluation keys honestly and sends them to the servers. A malicious client could construct malformed DPF keys such that the client's input gets counted more than once. To prevent this class of attacks, we propose a novel consistency check that only relies on inexpensive symmetric operations, like hashing.

We first ensure that the DPF output is non-zero only at a single point. The work of [94] introduces the primitive of verifiable DPF (VDPF). This is a stronger notion of DPF, where the servers obtain a correctness proof π upon evaluating a pair of DPF keys on a given input point. The two servers obtain the same proof π if the client generates the DPF keys honestly (i.e., the DPF output is non-zero only at a

single point α). Multiple proofs corresponding to different evaluation points are batch-verified. Next, we ensure that the DPF output value at the non-zero point is indeed 1. Our protocol instructs the servers to sum up all the output shares (corresponding to each point in $\mathbf{X}$) of the client and reconstruct the output. If the reconstructed output is not well-formed (i.e., is not 1), then the client's input is discarded. If the output is 1 (i.e., the client behaved honestly), then the DPF output shares are aggregated by the server in the histogram share.

Client Input Consistency Across Sessions. A malicious client can provide inconsistent inputs across the three server sessions by providing DPF keys for different points α_1, α_2, and α_3 in each session respectively. The verifiability of the VDPF fails to detect this attack since each individual VDPF in each session is valid.

To address the challenge, we propose a novel consistency check that relies on a single hash verification. Let us denote $\mathbf{Y}_{(0,1)}$, $\mathbf{Y}_{(0,2)}$, and $\mathbf{Y}_{(2,1)}$ be the output of the VDPF evaluation by $\mathcal{S}_0$ on keys $\mathsf{key}_{(0,1)}$, $\mathsf{key}_{(0,2)}$, and $\mathsf{key}_{(2,1)}$ corresponding to sessions $\mathcal{S}_0 - \mathcal{S}_1$, $\mathcal{S}_0 - \mathcal{S}_2$, and $\mathcal{S}_2 - \mathcal{S}_1$, respectively. Similarly, let us denote $\mathbf{Y}_{(1,0)}$, $\mathbf{Y}_{(2,0)}$, and $\mathbf{Y}_{(1,2)}$ be the output of the VDPF evaluation by $\mathcal{S}_1$ on keys $\mathsf{key}_{(1,0)}$, $\mathsf{key}_{(2,0)}$, and $\mathsf{key}_{(1,2)}$ corresponding to sessions $\mathcal{S}_0 - \mathcal{S}_1$, $\mathcal{S}_0 - \mathcal{S}_2$, and $\mathcal{S}_2 - \mathcal{S}_1$, respectively. By definition, reconstructing each pair of secret shared outputs (e.g., $\mathbf{Y}_{(0,1)}$, $\mathbf{Y}_{(1,0)}$) results in a vector of zeros except a single location. Note that the client has also sent $\mathsf{key}_{(2,1)}$ to $\mathcal{S}_0$ and $\mathsf{key}_{(2,0)}$ to $\mathcal{S}_1$ respectively. Server $\mathcal{S}_0$ sends hash $h \coloneqq \mathrm{H}(\mathbf{Y}_{(0,1)} - \mathbf{Y}_{(0,2)} \| \mathbf{Y}_{(0,2)} - \mathbf{Y}_{(2,1)})$ to $\mathcal{S}_1$, who verifies that $h = \mathrm{H}(\mathbf{Y}_{(2,0)} - \mathbf{Y}_{(1,0)} \| \mathbf{Y}_{(1,2)} - \mathbf{Y}_{(2,0)})$. The verification of the hash h ensures that the client's input is consistent between: (1) the sessions $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_0 - \mathcal{S}_2$, as well as (2) the sessions $\mathcal{S}_0 - \mathcal{S}_2$ and $\mathcal{S}_2 - \mathcal{S}_1$. By transitivity, all three sessions are consistent if the hash verification succeeds. Observe that if the servers acted honestly, $\mathbf{Y}_{(0,1)} + \mathbf{Y}_{(1,0)} = \mathbf{Y}_{(0,2)} + \mathbf{Y}_{(2,0)} = \mathbf{Y}_{(1,2)} + \mathbf{Y}_{(2,1)}$ and thus, $\mathbf{Y}_{(0,1)} - \mathbf{Y}_{(0,2)} = \mathbf{Y}_{(2,0)} - \mathbf{Y}_{(1,0)}$ and $\mathbf{Y}_{(0,2)} - \mathbf{Y}_{(2,1)} = \mathbf{Y}_{(1,2)} - \mathbf{Y}_{(2,0)}$. Our novel check requires additions (without any multiplications) and a cheap hash computation. The communication cost is one hash of size κ bits. This leads to $\mathcal{O}(\kappa\ell)$ server-server communication for ℓ clients, but it is optimized to logarithmic communication by applying batched client verification,

described in Section 7.5.

Limitations. The above histogram protocol is a building block for our heavy-hitters protocol and *is not* our final protocol. It suffers from the limitation that the client's input should lie in the subset $\mathbf{X}$ that the servers evaluate, i.e., $\alpha_i \in \mathbf{X}$ for $i \in [\ell]$. This leaks whether the client's input lies in $\mathbf{X}$ or not based on whether the evaluated DPF output in the consistency check is 0 or not. Our final histogram protocol addresses this issue by using techniques from Section 7.2.4, mainly replacing the VDPF with a VIDPF, and using the four consistency checks discussed in Section 7.2.4. The final histogram protocol is omitted for the sake of brevity.

7.2.3 Heavy-Hitters from $\mathcal{T}$-Prefix Count

Poplar reduced the problem of computing heavy hitters to the problem of computing prefix count queries for a given prefix $p \in \{0,1\}^*$ over client inputs. Then, they implemented prefix count queries by relying on incremental DPFs. However, their protocol leaks the count of strings that contain the $\mathcal{T}$ heavy-hitting prefix p due to the reliance on a prefix-count query oracle that outputs the exact count. To mitigate this leakage, we introduce the notion of $\mathcal{T}$-threshold prefix-count queries that return 1 if at least $\mathcal{T}$ of clients' input strings contain prefix p, otherwise, it returns 0. We define it as follows:

Definition 1 ($\mathcal{T}$-Prefix-count Query Oracle $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p,\mathcal{T})$)**.** *Return* 1 *(on input prefix* $p \in \{0,1\}^*$*) if prefix* p *appears at least* $\mathcal{T}$ *times in the clients' input strings* α_1*,* α_2*, …,* $\alpha_\ell \in \{0,1\}^*$ *where client* $\mathcal{C}_i$ *has input string* α_i *for* $i \in [\ell]$*, otherwise, return 0.*

$\mathcal{T}$-Heavy hitters. The $\mathcal{T}$-heavy hitters algorithm (for threshold $\mathcal{T}$) is provided with oracle $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p,\mathcal{T})$ for computing $\mathcal{T}$-prefix count for prefix p over the client input strings $\alpha_1,\ldots,\alpha_\ell$. The initial prefix is the empty string ϵ. At each level k, it considers the heavy-hitter prefixes $p \in \{0,1\}^k$ of length in set HH^k, which contains the list of k-bit strings that appear at least k times. The algorithm performs a breadth-first search of the prefix tree. It includes $k+1$ bit length strings $p \parallel 0$ in HH^{k+1} if $p \parallel 0$

occurs at least $\mathcal{T}$ times in the input strings $(\alpha_1, \ldots, \alpha_\ell)$, otherwise it gets pruned along its subtree. This is performed by querying the oracle $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 0, \mathcal{T})$. The same process is repeated for $p \parallel 1$. The algorithm repeats this for all k-bit strings in HH^k (which updates HH^{k+1} based on the search and pruning of set HH^k). At the end of the breadth-first search and pruning, the algorithm outputs the set of strings that are $\mathcal{T}$-heavy hitters. Our formal algorithm is presented in Fig. 7.3.

$\mathcal{T}$-Heavy Hitters from $\mathcal{T}$-prefix count queries

Parameters: Threshold $\mathcal{T} \in \mathbb{N}$ and string length $n \in \mathbb{N}$.

Inputs: The algorithm has no explicit input. It has access to t-prefix count query oracle $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p, t)$ for securely computing t-prefix-count queries over prefix p for strings $\alpha_1, \ldots, \alpha_\ell$.

Outputs: The set of $\mathcal{T}$-heavy-hitter strings in $\alpha_1, \alpha_2, \ldots, \alpha_\ell$.

Algorithm:

- Init. $\mathsf{HH}^{\leq n} := \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\} := \{\epsilon, \emptyset, \ldots, \emptyset\}$, where HH^0 contains empty string ϵ and $\mathsf{HH}^1, \ldots \mathsf{HH}^n$ are empty sets.
- For each prefix $p \in \mathsf{HH}^k$ of length k-bits in set HH^k (where $k = 0, 1, 2, \ldots n-1$) and $b \in \{0, 1\}$: If $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel b, \mathcal{T}) = 1$, $\mathsf{HH}^{k+1} := \mathsf{HH}^{k+1} \cup \{p \parallel b\}$.
- Output $\mathcal{T}$-heavy hitters $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\}$.

Figure 7.3: Algorithm for computing $\mathcal{T}$-heavy hitters.

Cost Analysis. There are ℓ input strings in total. For any string of length k, there are at most $\ell/\mathcal{T}$ candidate heavy hitter strings. At each level k, the algorithm makes at most one oracle query per heavy hitter string. Hence, the algorithm makes at most $n\ell/\mathcal{T}$ prefix-count oracle queries for n levels. If we set the threshold to be a constant fraction of all input strings (e.g., $\mathcal{T} = 0.01\ell$), then the number of prefix-count queries are independent of the number of input strings (e.g., $n\ell/\mathcal{T} = n\ell/0.01\ell = 100n$).

7.2.4 $\mathcal{T}$-Prefix Count Queries Oracle from VIDPF

We realize the $\mathcal{T}$-Prefix Count Query Oracle $\Omega(\cdot, \mathcal{T})$ from Def. 1 by relying on a new verifiable incremental DPF (VIDPF) primitive and using an ideal functionality $\mathcal{F}_{\mathsf{CMP}}$ (Fig. 7.10) for secure comparison.

7.2.4.1 Verifiable Incremental DPF (VIDPF)

A DPF allows a client to succinctly share a vector of size 2^n with a single non-zero point. Meanwhile, an incremental DPF (introduced by Poplar and denoted as IDPF) allows the client to succinctly secret share a path in the binary tree (used for representing 2^n leaves in binary format) and each node in the path can hold non-zero values. Our novel VIDPF primitive offers strong integrity guarantees over IDPFs since the evaluation of the client keys also provides proofs $(\pi_1, \ldots, \pi_n)$ to the servers ensuring that the VIDPF output is non-zero along a single path in the binary tree. It also allows incremental evaluation of the VIDPF over an input $x \in \{0,1\}^k$, given state st_b^{k-1} and proof π_b^{k-1}, corresponding to VIDPF evaluation of the first $k-1$ bits of x. The incremental evaluation enables the party possessing key_b to process one level and obtain the secret sharing of output $f(x)$, a new state st_b^k, and a new proof π_b^k corresponding to the VIDPF evaluation of the path involving x. More formally, we capture the high-level ideas of VIDPF using the following two algorithms:

- $\mathsf{Gen}(1^\kappa, 1^n, \alpha, (\beta^1, \beta^2, \ldots, \beta^n), \mathbb{G}) \rightarrow (\mathsf{key}_0, \mathsf{key}_1)$: Given security parameter κ, input size n, input string $\alpha \in \{0,1\}^n$, and values $\beta^1, \ldots, \beta^n$, the key generation algorithm outputs two VIDPF keys key_0 and key_1.

- $\mathsf{EvalPref}(b, \mathsf{key}_b, x, \mathsf{st}_b^{k-1}, \pi_b^{k-1}) \rightarrow (\mathsf{st}_b^k, y_b, \pi_b^k)$: Given a VIDPF key key_b and an input string $x \in \{0,1\}^k$ of length $k \leq n$ bits, the evaluation algorithm outputs an internal state st^k, secret-shared value $y_b \in \mathbb{G}$, and a proof $\pi_b^k \in \{0,1\}^*$.

Correctness of the VIDPF ensures that for all input points $\alpha \in \{0,1\}^n$, output values $\beta^1, \ldots, \beta^n \in \mathbb{G}$, VIDPF keys generated as $(\mathsf{key}_0, \mathsf{key}_1) \leftarrow \mathsf{Gen}(\alpha, \beta^1, \beta^2, \ldots, \beta^n, \mathbb{G})$ and all values $x \in \{0,1\}^k$, where $k \leq n$, the following holds for all $k \leq n$:

$$\pi_0^k = \pi_1^k \text{ and } y = (y_0 + y_1) = \begin{cases} \beta^k, \text{ if } x \text{ is a prefix of } \alpha, \\ 0, \text{ otherwise,} \end{cases}$$

where $(\mathsf{st}_0^k, y_0, \pi_0^k) \coloneqq \mathsf{EvalPref}(0, \mathsf{key}_0, x, \mathsf{st}_0^{k-1}, \pi_0^{k-1})$ and $(\mathsf{st}_1^k, y_1, \pi_1^k) \coloneqq \mathsf{EvalPref}(1, \mathsf{key}_1, x, \mathsf{st}_1^{k-1}, \pi_1^{k-1})$. For security guarantees, we require two additional properties from the VIDPF primitive:

- *Input Privacy.* The security of VIDPF guarantees that an adversarial evaluator in possession of either key_0 or key_1 (but not both), does not learn anything about the input α or the outputs $\beta^1, \ldots, \beta^n$ of the client.

- *Verifiability.* This property states that if two proofs (e.g., π_0^k and π_1^k) are the same, then there is at most one path of length k in the binary tree whose evaluation with $(\mathsf{key}_0, \mathsf{key}_1)$ outputs $(\beta^1, \beta^2, \ldots, \beta^k)$. More formally, for any $k \in [n]$ there exists a single k-bit string $\widetilde{x} \in \{0,1\}^k$ such that if $\pi_0^k = \pi_1^k$, then the following holds:

$$(\mathsf{st}_0^k, y_0, \pi_0^k) \coloneqq \mathsf{EvalPref}(0, \mathsf{key}_0, z, \mathsf{st}_0^{k-1}, \pi_0^{k-1})$$
$$(\mathsf{st}_1^k, y_1, \pi_1^k) \coloneqq \mathsf{EvalPref}(1, \mathsf{key}_1, z, \mathsf{st}_1^{k-1}, \pi_1^{k-1})$$
$$y_0 + y_1 = \begin{cases} \beta^k, \text{if } z = \widetilde{x}, \\ 0, \text{if } z = \{0,1\}^k \setminus \{\widetilde{x}\}, \end{cases}$$

  where $\mathsf{st}_0^{k-1}, \pi_0^{k-1}$ and $\mathsf{st}_1^{k-1}, \pi_1^{k-1}$ are obtained by running the $\mathsf{EvalPref}$ algorithm on $k-1$ bits of z. The evaluators initialize $\mathsf{st}_0^0 \coloneqq \mathsf{st}_1^0 \coloneqq 0$ and $\pi_0^0 \coloneqq \pi_1^0 \coloneqq 0$. It also implicitly captures the requirement that $\widetilde{x} \in \{0,1\}^{k-1}$ is a prefix of $\widetilde{x} \in \{0,1\}^k$ for $k \in [n]$.

We provide a construction of VIDPF in Figs. 7.4 and 7.5 based on length doubling PRG in the random oracle model. The security of our protocol is summarized in Theorem 3.

Theorem 3. *Assuming* $(\mathsf{PRG}, \mathsf{PRG}', \mathsf{PRG}'')$ *are pseudorandom generators, and* $(\mathrm{H}_1, \mathrm{H}_2)$ *are random oracles then* $\pi_{\mathsf{VIDPF}} = (\mathsf{Gen}, \mathsf{EvalPref})$ *in Figs. 7.4 and 7.5 is a VIDPF.*

Proof. Input privacy of our VIDPF follows from the input privacy of the underlying IDPF protocol from Poplar, which in turn relies on the pseudorandomness of PRG.

Adding $\mathsf{cs}^{(i)}$ in steps 16-17 does not affect the input privacy of the client in the random oracle model since $\mathsf{cs}^{(i)} = \widetilde{\pi}_0^{(i)} \oplus \widetilde{\pi}_1^{(i)}$ is an XOR of two random oracle outputs. Each server will know the preimage of either $\widetilde{\pi}_0^{(i)}$ or the preimage of $\widetilde{\pi}_1^{(i)}$ by evaluating the given VIDPF key. The server breaks input privacy if it computes both preimages. However, to compute the other preimage it needs to invert the random oracle on $\widetilde{\pi}_{1-b'}^{(i)}$ (assuming it obtained the preimage of $\widetilde{\pi}_{b'}^{(i)}$ by evaluating the VIDPF key).

A malicious client breaks the verifiability property if there are two non-zero paths, say u and v in the evaluation tree such that the client still passes the verification check performed by the servers on $\mathsf{cs}^{(i)}$. This means the servers obtain $s_0^i(u)$, $s_1^i(u)$, $s_0^i(v)$ and $s_1^i(v)$ from Step 11 of EvalNext (Fig. 7.5) by evaluating on u and v such that the following holds:

$$s_0^i(u) \neq s_1^i(u) \text{ and } s_0^i(v) \neq s_1^i(v)$$

$$\mathsf{cs}^{(i)} = \widetilde{\pi}_0^{(i)}(u) \oplus \widetilde{\pi}_1^{(i)}(u) = \widetilde{\pi}_0^{(i)}(v) \oplus \widetilde{\pi}_1^{(i)}(v),$$

where $\widetilde{\pi}_b^{(i)}(u) \coloneqq \mathrm{H}_1(u, s_0^i(u))$ and $\widetilde{\pi}_b^{(i)}(v) \coloneqq \mathrm{H}_1(v, s_0^i(v))$ for $b \in \{0, 1\}$. However, this is not possible in the random oracle model since it breaks the XOR-collision-resistance property of the random oracle H_1. The adversary cannot find such a set of $s_0^i(u)$, $s_1^i(u)$, $s_0^i(v)$ and $s_1^i(v)$ values. Lemma 3 of [94] captures the formal details. In addition, we also rely on the collision resistance property of H_2 for arguing verifiability when multiple proofs are iteratively hashed together in step 12 of the EvalNext algorithm. □

Next, we outline our protocol for securely implementing $\mathcal{T}$-prefix count queries using VIDPF and the comparison functionality $\mathcal{F}_{\mathsf{CMP}}$.

7.2.4.2 Implementing $\mathcal{T}$-Prefix Count Queries

Each client generates three pairs of VIDPF keys, one for each pair of servers, with independent randomness but the same input point α and output values $(1, \ldots, 1)$. The client sends the keys for the sessions to the respective servers (Fig. 7.1) as in our histogram protocol.

Notation: We denote the private n-bit string α and its bit decomposition as $\alpha_1, \ldots, \alpha_n \in \{0,1\}^n$.
Primitives: $\mathsf{PRG} : \{0,1\}^\kappa \to \{0,1\}^{2\kappa+2}$ is a pseudorandom generator. $\mathrm{H}_1 : \{0,1\}^* \times \{0,1\}^\kappa \to \{0,1\}^{2\kappa}$ and $\mathrm{H}_2 : \{0,1\}^{2\kappa} \to \{0,1\}^{2\kappa}$ are random oracles.

Gen$(1^\kappa, 1^n, \alpha, (\beta_1, \beta_2, \ldots \beta_n), \mathbb{G})$**:** ▷ Generate DPF keys.

1: Sample $s_b^{(0)} \leftarrow \{0,1\}^\kappa$ for $b \in \{0,1\}$ ▷ Secret seeds.
2: Let $t_0^{(0)} := 0$ and $t_1^{(0)} := 1$
3: **for** $i := 1$ to n **do** ▷ For each bit of α.
4: $s_b^L \| t_b^L \| s_b^R \| t_b^R := \mathsf{PRG}(s_b^{(i-1)})$ for $b \in \{0,1\}$ ▷ Parse the output of PRG as a sequence of $(\kappa \| 1 \| \kappa \| 1)$ bits.
5: **if** $\alpha_i = 0$ **then** $\mathsf{Diff} := L$, $\mathsf{Same} := R$ ▷ Set right children to be equal.
6: **else** $\mathsf{Diff} := R$, $\mathsf{Same} := L$ ▷ Set left children to be equal.
7: $s_{\mathsf{cw}} := s_0^{\mathsf{Same}} \oplus s_1^{\mathsf{Same}}$
8: $t_{\mathsf{cw}}^L := t_0^L \oplus t_1^L \oplus \alpha_i \oplus 1$ ▷ Left control bits not equal if $\alpha_i = 0$.
9: $t_{\mathsf{cw}}^R := t_0^R \oplus t_1^R \oplus \alpha_i$ ▷ Right control bits not equal if $\alpha_i = 1$.
10: $\tilde{s}_b^{(i)} := s_b^{\mathsf{Diff}} \oplus t_b^{(i-1)} \cdot s_{\mathsf{cw}}$ for $b \in \{0,1\}$ ▷ Correction.
11: $t_b^{(i)} := t_b^{\mathsf{Diff}} \oplus t_b^{(i-1)} \cdot t_{\mathsf{cw}}^{\mathsf{Diff}}$ for $b \in \{0,1\}$ ▷ Correction.
12: $s_b^{(i)} \| W_b^{(i)} := \mathsf{Convert}(\tilde{s}_b^{(i)})$ for $b \in \{0,1\}$
13: $W_{\mathsf{cw}}^{(i)} := (-1)^{t_1^{(i)}} \cdot [\beta_i - W_0^{(i)} + W_1^{(i)}]$ ▷ Output correction.
14: $\mathsf{cw}^{(i)} := s_{\mathsf{cw}} \| t_{\mathsf{cw}}^L \| t_{\mathsf{cw}}^R \| W_{\mathsf{cw}}^{(i)}$ ▷ Correction word for level i.
15: $\widetilde{\pi}_b^{(i)} = \mathrm{H}_1(\alpha_{\le i} \| s_b^{(i)})$
16: $\mathsf{cs}^{(i)} = \widetilde{\pi}_0^{(i)} \oplus \widetilde{\pi}_1^{(i)}$.
17: $\mathsf{key}_b := (s_b^{(0)} \| \mathsf{cw}^{(1)} \| \ldots \| \mathsf{cw}^{(n)} \| \mathsf{cs}^{(1)} \| \ldots \| \mathsf{cs}^{(n)})$ for $b \in \{0,1\}$ ▷ Key for party b.
18: **return** key_b for $b \in \{0,1\}$

Convert$_{\mathbb{G}}(s)$**:**

1: Let $u \leftarrow |\mathbb{G}|$.
2: **if** $u = 2^m$ for an integer m **then:**
3: Return the group element represented by $\mathsf{PRG}'(s) \mod u$,
4: where $\mathsf{PRG}' : \{0,1\}^\kappa \to \{0,1\}^m$.
5: **else:**
6: Let $n = \lceil \log_2 u \rceil + \kappa$.
7: Return the group element represented by $\mathsf{PRG}''(s) \mod u$,
8: where $\mathsf{PRG}' : \{0,1\}^\kappa \to \{0,1\}^n$.

Figure 7.4: Protocol π_{VIDPF} for Verifiable Incremental DPF (continues in Fig. 7.5).

EvalNext$(b, i, \mathsf{st}^{(i-1)}, \mathsf{cw}^{(i)}, \mathsf{cs}^{(i)}, x_{\le i}, \pi)$: ▷ Evaluate x_i.

1: Parse $\mathsf{st}^{(i-1)}$ as $(s^{i-1} \parallel t^{i-1})$.

2: $s_{\mathsf{cw}} \parallel t^L_{\mathsf{cw}} \parallel t^R_{\mathsf{cw}} \parallel W^{(i)}_{\mathsf{cw}} := \mathsf{cw}^i$ ▷ Parse correction word.

3: $\tilde{s}^L \parallel \tilde{t}^L \parallel \tilde{s}^R \parallel \tilde{t}^R := \mathsf{PRG}(s^{(i-1)})$ ▷ Parse the output of PRG as a sequence of $(\kappa \parallel 1 \parallel \kappa \parallel 1)$ bits.

4: $\tau^{(i)} := (\tilde{s}^L \parallel \tilde{t}^L \parallel \tilde{s}^R \parallel \tilde{t}^R) \oplus (t^{(i-1)} \cdot [s_{\mathsf{cw}} \parallel t^L_{\mathsf{cw}} \parallel s_{\mathsf{cw}} \parallel t^R_{\mathsf{cw}}])$

5: $s^L \parallel t^L \parallel s^R \parallel t^R := \tau^{(i)}$ ▷ Parse $\tau^{(i)}$.

6: **if** $x_i = 0$ **then** $\tilde{s}^{(i)} := s^L, \quad t^{(i)} := t^L$ ▷ Keep left path.

7: **else** $\tilde{s}^{(i)} := s^R, \quad t^{(i)} := t^R$ ▷ Keep right path.

8: $s^{(i)} \parallel W^{(i)} := \mathsf{Convert}(\tilde{s}^{(i)})$ ▷ New seed and output for level i.

9: $\mathsf{st}^{(i)} := s^{(i)} \parallel t^{(i)}$ ▷ Save the state.

10: $y^{(i)} := (-1)^b \cdot [W^{(i)} + t^{(i)} \cdot W_{\mathsf{cw}}]$ ▷ Compute output at level i.

11: $\widetilde{\pi}^{(i)} = \mathrm{H}_1(x^{\le i} \parallel s^{(i)})$.

12: $\pi = \pi \oplus \mathrm{H}_2(\pi \oplus (\widetilde{\pi}^{(i)} \oplus t^{(i)} \cdot \mathsf{cs}^{(i)}))$.

13: **return** $(\mathsf{st}^{(i)}, y^{(i)}, \pi)$

EvalPref$(b, \mathsf{key}, x \in \{0,1\}^n, \mathsf{st}^{(d-1)}, d, \pi)$: ▷ Evaluate one public bitstring x on all it's bits x_i for $i \in [n]$.

1: Parse key as $s^{(0)} \parallel \mathsf{cw}^{(1)} \parallel \ldots \parallel \mathsf{cw}^{(n)} \parallel \mathsf{cs}^{(1)} \parallel \ldots \parallel \mathsf{cs}^{(n)}$. ▷ Parse key for party b.

2: **if** $(d \neq 1)$ **then** parse $\mathsf{st}^{(d-1)}$ as $(s^{(d-1)} \parallel t^{(d-1)})$,

3: **else** $t^{(0)} := b, \quad \mathsf{st}^{(0)} := s^{(0)} \parallel t^{(0)}$.

4: **for** $i := d$ to n **do** ▷ For each bit of x.

5: $\quad (\mathsf{st}^{(i)}, y^{(i)}, \pi) := \mathsf{EvalNext}(b, i, \mathsf{st}^{(i-1)}, \mathsf{cw}^i, \mathsf{cs}^i, x^{\le i}, \pi)$.

6: **return** $(\mathsf{st}^{(n)}, y^{(n)}, \pi)$

Figure 7.5: Protocol π_{VIDPF} for Verifiable Incremental DPF (continuing from Fig. 7.4).

Basic Protocol. As depicted in Fig. 7.2, $\mathcal{S}_1$ replicates $\mathcal{S}_2$ in the $\mathcal{S}_2 - \mathcal{S}_0$ session and $\mathcal{S}_2$ behaves as an attestator for $\mathcal{S}_1$ by sending hashes of the messages that $\mathcal{S}_1$ should send. The hash prevents server $\mathcal{S}_1$ from acting maliciously corresponding to the $\mathcal{S}_2 - \mathcal{S}_0$ session. Similar protocol steps are run by $\mathcal{S}_2$ for the session $\mathcal{S}_1 - \mathcal{S}_2$, where $\mathcal{S}_2$ sends hashes to $\mathcal{S}_1$. Hence, $\mathcal{S}_0$ and $\mathcal{S}_1$ run three sessions, and $\mathcal{S}_2$ runs two of those sessions in parallel. Next, we describe the protocol to compute a $\mathcal{T}$-prefix count query on a string $p \parallel 0 \in \{0,1\}^k$ (note, the same process can be repeated for query string $p \parallel 1$). The servers $\mathcal{S}_0$ and $\mathcal{S}_1$ evaluate the VIDPF keys for the three sessions on $p \parallel 0$ and obtain a secret share of the output $y^{p\|0}$ and proof π. Ideally, $y^{p\|0}$ should be $\beta^k = 1$ for an honest client. However, a malicious client could construct malformed VIDPF keys such that the client's input gets counted more than once.

Client Input Validation. We introduce the following consistency checks to validate a client's input. Checks 1-3 ensure that the VIDPF keys are *"one-hot"*, i.e., they have a single non-zero evaluation path (containing 1 in this case, along the path), and check 4 ensures that the client input is consistent across the sessions:

- *Check 1:* The servers $\mathcal{S}_0$ and $\mathcal{S}_1$ first verify that the proofs π are the same for all three sessions. This ensures that there is at most one path in the binary tree that is non-zero.

- *Check 2:* For the root level (i.e., $k = 0$), the servers evaluate the VIDPF keys on the empty string ϵ and verify it is 1.

- *Check 3:* Finally, at the k^{th} level, the servers need to verify that $y^{p\|0}$ is either 0 or 1, without reconstructing the output. We perform this check by observing that the output of the parent p should be the sum of the outputs of $p \parallel 0$ and $p \parallel 1$. The servers evaluate the VIDPF keys on the parent string p and sibling (of $p \parallel 0$) string $p \parallel 1$ to obtain secret shares of the output of y^p and $y^{p\|1}$ respectively. The servers reconstruct $y^p - (y^{p\|0} + y^{p\|1})$ and verify that it is 0. The first check ensures that at most one of $y^{p\|0}$ or $y^{p\|1}$ is non-zero. Combining the two checks, we can

conclude that either $(y^{p\|0} = 0, y^{p\|1} = 1)$ or $(y^{p\|0} = 1, y^{p\|1} = 0)$, since at most one child can equal 1 when the parent holds a value of 1. Iterating this for all k levels ensures that $y^{p\|0} = 1$ if and only if $y^p = 1$ and $y^{p\|1} = 0$, else $y^{p\|0} = 0$. The servers also verify (using check 1) the corresponding proofs π generated during the VIDPF evaluation along the path, to ensure there is at most one non-zero path in the entire binary tree.

- *Check 4:* The servers also need to ensure that the client input is consistent across the three server sessions. This is ensured by computing the difference of the reconstructed outputs across the sessions and verifying that they are equal to 0 by matching their hash values. For more details, we defer to Section 7.3.

Output Phase. Once the client's VIDPF output $y^{p\|0}$ is verified, the secret shares of $y^{p\|0}$ are aggregated into counter $\mathsf{cnt}^{p\|0}$. The servers repeat the above steps for all the clients in parallel to obtain secret shares of $\mathsf{cnt}^{p\|0}$. The servers invoke the comparison functionality $\mathcal{F}_{\mathsf{CMP}}$ (Fig. 7.10) with the secret shares of cnt and threshold $\mathcal{T}$. $\mathcal{F}_{\mathsf{CMP}}$ reconstructs cnt and it outputs 1 if $\mathsf{cnt} \geq \mathcal{T}$, otherwise, it outputs 0. This is returned by the servers as the output of the $\mathcal{T}$-prefix count oracle query response to the string $p \| 0$. Similar steps are run for $p \| 1$. The comparison functionality $\mathcal{F}_{\mathsf{CMP}}$ is securely implemented using the state-of-the-art protocol of Rabbit [171].

Robustness Against a Malicious Server. Note that the above validation check assumes that both servers are honest. Otherwise, malicious behaviour is detected as described next. The third server ensures that if the client behaves honestly then at least one of the three sessions will be evaluated correctly since two of the servers are honest. After aggregating all the client's inputs, cnt is reconstructed across the three sessions by $\mathcal{F}_{\mathsf{CMP}}$. If cnt is inconsistent across any pair of servers then $\mathcal{F}_{\mathsf{CMP}}$ returns $\perp$ indicating that one of the servers behaved maliciously by launching an additive attack. This causes the honest servers to abort, providing robustness against the malicious server. We observe that our protocol satisfies *fairness* (which is a stronger security notion than selective abort) if $\mathcal{F}_{\mathsf{CMP}}$ is implemented using a fair protocol.

Batched Client Verification. In our final protocol, we verify multiple client inputs at each level in one batch. We batch all the clients' VIDPF evaluations using a Merkle tree that has ℓ leaves for ℓ clients. First, the servers check the equality of ℓ leaves by asserting that the Merkle roots are the same. If the roots match then the leaves are the same, while if they differ then the servers recursively repeat the same process for each of the two children of the parent node. Proceeding this way, the servers identify the malformed leaves on which the two trees differ. This reduces the dependency of our server-to-server communication to $\mathcal{O}(\ell'(\log_2 \frac{\ell}{\ell'}))$, for ℓ' malicious clients, instead of $\mathcal{O}(\ell)$, while when $\ell' = 0$ our communication is down to $\mathcal{O}(1)$. Formal details can be found in Section 7.5.

Functionality $\mathcal{F}_{\mathsf{HH}}$

Parameters: Servers $\mathcal{S}_0$, $\mathcal{S}_1$, and $\mathcal{S}_2$, and ℓ clients $\mathcal{C}_i$ for $i \in [\ell]$. Servers $\mathcal{S}_0$, $\mathcal{S}_1$, and $\mathcal{S}_2$ agree upon:

- A bound ℓ on the number of client submissions.
- A bound $\mathcal{T}$ on the threshold for heavy hitters.

Inputs: *Servers* $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$ do not have any input. *Clients* $\mathcal{C}_i$: A point $\alpha_i \in \{0,1\}^n$ for $i \in [\ell]$. $\alpha_{i,j}$ represents the jth bit of α_i.

Outputs: Init. $\mathsf{HH}^{\leq n} \coloneqq \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\} \coloneqq \{\epsilon, \emptyset, \ldots, \emptyset\}$. Repeat for length of k bits, where $k \in [0, \ldots, n-1]$ and for each prefix $p \in \mathsf{HH}^k$:

- Update $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup (p \parallel b)$ if

$$\textstyle\sum_{i=1}^{\ell} \left|(\alpha_{i,\leq k+1} = (p \parallel b))\right| \geq \mathcal{T}, \text{ for } b \in \{0,1\}.$$

$\mathcal{F}_{\mathsf{HH}}$ outputs the following:

- *Servers* $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$: Set of $\mathcal{T}$-heavy hitters $\mathsf{HH}^{\leq n}$.
- *Clients* $\mathcal{C}_i$: No output for $i \in [\ell]$.

Corruption: Adversary $\mathcal{A}$ maliciously corrupts one server and multiple clients together. $\mathcal{A}$ can perform the following:

- If $\mathcal{A}$ instructs the functionality to discard the jth client's input then $\mathcal{F}_{\mathsf{HH}}$ discards α_j from the output computation.
- If $\mathcal{A}$ instructs the functionality to abort at level $k+1$ by sending $(\perp, k+1)$, then the functionality returns $\mathsf{HH}^{\leq k}$ to $\mathcal{A}$ and the honest servers; additionally, the functionality instructs the honest servers to abort by sending $\perp$.

Figure 7.6: The ideal $\mathcal{F}_{\mathsf{HH}}$ functionality for $\mathcal{T}$-heavy hitters.

7.3 Private Heavy Hitters

We provide the ideal functionality $\mathcal{F}_{\mathsf{HH}}$ for heavy-hitters between three servers and ℓ clients in Fig. 7.6. Adversary $\mathcal{A}$ maliciously corrupts any one of the servers and multiple clients. Note that this corruption can easily happen; if $\mathcal{A}$ has maliciously corrupt a server, then $\mathcal{A}$ can spawn multiple malicious clients. Additionally, if $\mathcal{A}$ controls a server, it can instruct $\mathcal{F}_{\mathsf{HH}}$ to discard an honest client's input. It can also instruct the functionality to abort at a particular level $k+1$. In this case, $\mathcal{A}$ and the honest servers receive the set of all (that have not been discarded by $\mathcal{A}$) k-bit heavy-hitting prefixes as output, and the functionality instructs the honest servers to abort. We remark that $\mathcal{F}_{\mathsf{HH}}$ never leaks the honest client's inputs.

Our detailed protocol π_{HH} that implements $\mathcal{F}_{\mathsf{HH}}$ appears in Figs. 7.7 and 7.8, while high-level ideas of our protocol can be found in Sections 7.2.3 and 7.2.4. Our π_{HH} protocol privately computes all the $\mathcal{T}$-heavy-hitting strings (and their heavy-hitting prefixes) given the input data of ℓ clients, while protecting the privacy of the individual data points. π_{HH} runs on three servers $(\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2)$ that utilize our verifiable incremental DPF (VIDPF) protocol to privately aggregate the clients' data points. Specifically, π_{HH} runs three VIDPF sessions, which guarantees security against a malicious server. Our protocol proceeds in three phases: a client computation phase, a server computation phase, and an output phase.

Client Computation. During the client computation phase, each client $\mathcal{C}$ prepares three pairs of VIDPF keys for their private data point $\alpha \in \mathbf{X}$, and output value $(\beta^1, \ldots, \beta^n) := (1, \ldots, 1)$ along the path to α, using independent randomness for each key generation. Employing three pairs of keys essentially allows us to run three separate VIDPF sessions. $\mathcal{S}_0$ and $\mathcal{S}_1$ each have one key for each of the three sessions, while $\mathcal{S}_2$ acts as a consistency checking server and shares one key with each of the other two servers. More specifically, the client generates $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(0,2)})$ for $\mathcal{S}_0$, $(\mathsf{key}_{(1,0)}, \mathsf{key}_{(1,2)})$ for $\mathcal{S}_1$, and $(\mathsf{key}_{(2,1)}, \mathsf{key}_{(2,0)})$ for $\mathcal{S}_2$. The client sends $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(0,2)}, \mathsf{key}_{(2,1)})$ to $\mathcal{S}_0$, $(\mathsf{key}_{(1,0)}, \mathsf{key}_{(1,2)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_1$, and $(\mathsf{key}_{(2,1)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_2$ as shown in Fig. 7.1.

- **Input:** Each client $\mathcal{C}_i$ has an input point $\alpha_i \in \mathbf{X}$ for $i \in [\ell]$.
- **Output:** The servers $\mathcal{S}_b$ (for $b \in \{0,1,2\}$) output the set of $\mathcal{T}$-heavy hitters $\mathsf{HH}^{\leq n} := \mathcal{F}_{\mathsf{HH}}(\ell, \mathcal{T}, \{\alpha_i\}_{i\in[\ell]})$.
- **Primitive:** VIDPF $:= (\mathsf{Gen}, \mathsf{EvalPref}, \mathsf{EvalNext})$ is a verifiable incremental DPF. $\mathrm{H}_1, \mathrm{H}_2 : \{0,1\}^* \to \{0,1\}^\kappa$ are random oracles.

Client $\mathcal{C}$ Computation. **(Repeated for ℓ clients, each of which has their own private input α)**

1. Client $\mathcal{C}$ with input α prepares three pairs DPF keys with independent randomness $u, v, w \leftarrow \{0,1\}^\kappa$, as: $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(1,0)}) := \mathsf{Gen}(1^\kappa, 1^n, \alpha, (1,\ldots,1), \mathbb{G}), (\mathsf{key}_{(1,2)}, \mathsf{key}_{(2,1)}) := \mathsf{Gen}(1^\kappa, 1^n, \alpha, (1,\ldots,1), \mathbb{G}), (\mathsf{key}_{(2,0)}, \mathsf{key}_{(0,2)}) \coloneqq \mathsf{Gen}(1^\kappa, 1^n, \alpha, (1,\ldots,1), \mathbb{G})$
2. The client sends $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(0,2)}, \mathsf{key}_{(2,1)})$ to $\mathcal{S}_0$, $(\mathsf{key}_{(1,0)}, \mathsf{key}_{(1,2)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_1$ and $(\mathsf{key}_{(2,1)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_2$.

Server Computation.

The servers initialize $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\} := \{\epsilon, \emptyset, \ldots, \emptyset\}$, where HH^0 contains empty string ϵ and $\mathsf{HH}^1, \ldots \mathsf{HH}^n$ are empty sets.

For $k \in [0, \ldots, n-1]$ repeat the following steps: ▷ n is the number of bits.

1. **Initialization.** For prefix $p \in \mathsf{HH}_b^k$, servers initialize the aggregation variables for prefixes $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ as follows: $\mathcal{S}_0$ sets $\mathsf{cnt}^\gamma_{(0,1)} := \mathsf{cnt}^\gamma_{(0,2)} := \mathsf{cnt}^\gamma_{(2,1)} := 0$, $\mathcal{S}_1$ sets $\mathsf{cnt}^\gamma_{(1,2)} := \mathsf{cnt}^\gamma_{(1,0)} := \mathsf{cnt}^\gamma_{(2,0)} := 0$, and $\mathcal{S}_2$ sets $\mathsf{cnt}^\gamma_{(2,0)} := \mathsf{cnt}^\gamma_{(2,1)} := 0$.
2. **VIDPF Evaluation.** For prefix $p \in \mathsf{HH}_b^{\leq k}$, Server $\mathcal{S}_b$ computes: **(Repeated for ℓ clients)**

   (a) **If** $p = \emptyset$ **then** $\mathcal{S}_0$ sets $\mathsf{st}^\emptyset_{(0,1)} := \pi^\emptyset_{(0,1)} := \mathsf{st}^\emptyset_{(0,2)} := \pi^\emptyset_{(0,2)} := \mathsf{st}^\emptyset_{(2,1)} := \pi^\emptyset_{(2,1)} := \emptyset$, $\mathcal{S}_1$ sets $\mathsf{st}^\emptyset_{(1,2)} := \pi^\emptyset_{(1,2)} := \mathsf{st}^\emptyset_{(1,0)} := \pi^\emptyset_{(1,0)} := \mathsf{st}^\emptyset_{(2,0)} := \pi^\emptyset_{(2,0)} := \emptyset$. $\mathcal{S}_2$ sets $\mathsf{st}^\emptyset_{(2,0)} := \pi^\emptyset_{(2,0)} := \mathsf{st}^\emptyset_{(2,1)} := \pi^\emptyset_{(2,1)} := \emptyset$.

   If $p \neq \emptyset$ **then** each server $\mathcal{S}_b$ retrieves the following states from memory corresponding to the internal states of π_{VIDPF} computation for prefix p: $\mathcal{S}_0$ retrieves $(\mathsf{st}^p_{(0,1)}, y^p_{(0,1)}, \pi^p_{(0,1)})$, $(\mathsf{st}^p_{(0,2)}, y^p_{(0,2)}, \pi^p_{(0,2)})$ and $(\mathsf{st}^p_{(2,1)}, y^p_{(2,1)}, \pi^p_{(2,1)})$. $\mathcal{S}_1$ retrieves $(\mathsf{st}^p_{(1,2)}, y^p_{(1,2)}, \pi^p_{(1,2)})$, $(\mathsf{st}^p_{(1,0)}, y^p_{(1,0)}, \pi^p_{(1,0)})$ and $(\mathsf{st}^p_{(2,0)}, y^p_{(2,0)}, \pi^p_{(2,0)})$. $\mathcal{S}_2$ retrieves $(\mathsf{st}^p_{(2,0)}, y^p_{(2,0)}, \pi^p_{(2,0)})$ and $(\mathsf{st}^p_{(2,1)}, y^p_{(2,1)}, \pi^p_{(2,1)})$.

   (b) Each server $\mathcal{S}_b$ evaluates the VIDPF on prefixes $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ and stores them in memory:

   $\mathcal{S}_0$ computes $(\mathsf{st}^\gamma_{(0,1)}, y^\gamma_{(0,1)}, \pi^\gamma_{(0,1)}) := \mathsf{EvalPref}(0, \mathsf{key}_{(0,1)}, \gamma, \mathsf{st}^p_{(0,1)}, k, \pi^p_{(0,1)})$ and stores it.

   $\mathcal{S}_0$ computes $(\mathsf{st}^\gamma_{(0,2)}, y^\gamma_{(0,2)}, \pi^\gamma_{(0,2)}) := \mathsf{EvalPref}(1, \mathsf{key}_{(0,2)}, \gamma, \mathsf{st}^p_{(0,2)}, k, \pi^p_{(0,2)})$ and stores it.

   $\mathcal{S}_1$ computes $(\mathsf{st}^\gamma_{(1,2)}, y^\gamma_{(1,2)}, \pi^\gamma_{(1,2)}) := \mathsf{EvalPref}(0, \mathsf{key}_{(1,2)}, \gamma, \mathsf{st}^p_{(1,2)}, k, \pi^p_{(1,2)})$ and stores it.

   $\mathcal{S}_1$ computes $(\mathsf{st}^\gamma_{(1,0)}, y^\gamma_{(1,0)}, \pi^\gamma_{(1,0)}) := \mathsf{EvalPref}(1, \mathsf{key}_{(1,0)}, \gamma, \mathsf{st}^p_{(1,0)}, k, \pi^p_{(1,0)})$ and stores it.

   $\mathcal{S}_2$ and $\mathcal{S}_1$ compute $(\mathsf{st}^\gamma_{(2,0)}, y^\gamma_{(2,0)}, \pi^\gamma_{(2,0)}) := \mathsf{EvalPref}(0, \mathsf{key}_{(2,0)}, \gamma, \mathsf{st}^p_{(2,0)}, k, \pi^p_{(2,0)})$ and store them.

   $\mathcal{S}_2$ and $\mathcal{S}_0$ compute $(\mathsf{st}^\gamma_{(2,1)}, y^\gamma_{(2,1)}, \pi^\gamma_{(2,1)}) := \mathsf{EvalPref}(1, \mathsf{key}_{(2,1)}, \gamma, \mathsf{st}^p_{(2,1)}, k, \pi^p_{(2,1)})$ and store them.

   (c) **If** $k = 1$ **then** the servers compute the proof that the VIDPF evaluation at the root sums up to 1:

   $\mathcal{S}_0$ sets $h^\emptyset_{(0,1)} := \mathrm{H}_1(\emptyset, 1 - y^0_{(0,1)} - y^1_{(0,1)})$ and $h^\emptyset_{(0,2)} := \mathrm{H}_1(\emptyset, y^0_{(0,2)} + y^1_{(0,2)},)$,

   $\mathcal{S}_1$ sets $h^\emptyset_{(1,2)} := \mathrm{H}_1(\emptyset, 1 - y^0_{(1,2)} - y^1_{(1,2)})$ and $h^\emptyset_{(1,0)} := \mathrm{H}_1(\emptyset, y^0_{(1,0)} + y^1_{(1,0)})$,

   $\mathcal{S}_2$ and $\mathcal{S}_1$ set $h^\emptyset_{(2,0)} := \mathrm{H}_1(\emptyset, 1 - y^0_{(2,0)} - y^1_{(2,0)})$, $\mathcal{S}_2$ and $\mathcal{S}_0$ set $h^\emptyset_{(2,1)} := \mathrm{H}_1(\emptyset, y^0_{(2,1)} - y^1_{(2,1)})$.

Figure 7.7: **Private $\mathcal{T}$-Heavy Hitters Protocol π_{HH}** (continues in Fig. 7.8).

2. (c) **VIDPF Evaluation.** ▷ (Continued from Fig. 7.7)

If $k \neq 1$ **then** the servers compute the proof that (VIDPF output on prefix p) = (VIDPF output on prefix $p \,\|\, 0$) + (VIDPF output on prefix $p \,\|\, 1$):

$$\mathcal{S}_0 \text{ computes } h^p_{(0,1)} \coloneqq \mathrm{H}_1(p, y^p_{(0,1)} - y^{p\|0}_{(0,1)} - y^{p\|1}_{(0,1)}) \text{ and } h^p_{(0,2)} \coloneqq \mathrm{H}_1(p, -(y^p_{(0,2)} - y^{p\|0}_{(0,2)} - y^{p\|1}_{(0,2)}))$$

$$\mathcal{S}_1 \text{ computes } h^p_{(1,2)} \coloneqq \mathrm{H}_1(p, y^p_{(1,2)} - y^{p\|0}_{(1,2)} - y^{p\|1}_{(1,2)}) \text{ and } h^p_{(1,0)} \coloneqq \mathrm{H}_1(p, -(y^p_{(1,0)} - y^{p\|0}_{(1,0)} - y^{p\|1}_{(1,0)}))$$

$$\mathcal{S}_2 \text{ and } \mathcal{S}_1 \text{ compute } h^p_{(2,0)} \coloneqq \mathrm{H}_1(p, y^p_{(2,0)} - y^{p\|0}_{(2,0)} - y^{p\|1}_{(2,0)}),$$

$$\mathcal{S}_2 \text{ and } \mathcal{S}_0 \text{ compute } h^p_{(2,1)} \coloneqq \mathrm{H}_1(p, -(y^p_{(2,1)} - y^{p\|0}_{(2,1)} - y^{p\|1}_{(2,1)})).$$

(d) $\mathcal{S}_0$ and $\mathcal{S}_1$ ensure that the client input is consistent across the sessions:

$$\mathcal{S}_0 \text{ computes } \widehat{h^{p\|0}} = \mathrm{H}_1(y^{p\|0}_{(0,1)} - y^{p\|0}_{(0,2)}, y^{p\|0}_{(0,2)} - y^{p\|0}_{(2,1)}) \text{ and } \widehat{h^{p\|1}} = \mathrm{H}_1(y^{p\|1}_{(0,1)} - y^{p\|1}_{(0,2)}, y^{p\|1}_{(0,2)} - y^{p\|1}_{(2,1)}).$$

$$\mathcal{S}_1 \text{ computes } \overline{h^{p\|0}} \coloneqq \mathrm{H}_1(y^{p\|0}_{(2,0)} - y^{p\|0}_{(1,0)}, y^{p\|0}_{(1,2)} - y^{p\|0}_{(2,0)})) \text{ and } \overline{h^{p\|1}} \coloneqq \mathrm{H}_1(y^{p\|1}_{(2,0)} - y^{p\|1}_{(1,0)}, y^{p\|1}_{(1,2)} - y^{p\|1}_{(2,0)}))$$

(e) *Client State Accumulation:* The servers accumulate their local state for each client session:

$$\mathcal{S}_0 \text{ sets } R^k_{(0,1)} \coloneqq \mathrm{H}_2\big(||_{p\in\mathsf{HH}^k}\,(p, h^p_{(0,1)}, \pi^{p\|0}_{(0,1)}, \pi^{p\|1}_{(0,1)})\big) \text{ and } R^k_{(0,2)} \coloneqq \mathrm{H}_2\big(||_{p\in\mathsf{HH}^k}\,(p, h^p_{(0,2)}, \pi^{p\|0}_{(0,2)}, \pi^{p\|1}_{(0,2)})\big)$$

$$\mathcal{S}_1 \text{ sets } R^k_{(1,2)} \coloneqq \mathrm{H}_2\big(||_{p\in\mathsf{HH}^k}\,(p, h^p_{(1,2)}, \pi^{p\|0}_{(1,2)}, \pi^{p\|1}_{(1,2)})\big) \text{ and } R^k_{(1,0)} \coloneqq \mathrm{H}_2\big(||_{p\in\mathsf{HH}^k}\,(p, h^p_{(1,0)}, \pi^{p\|0}_{(1,0)}, \pi^{p\|1}_{(1,0)})\big)$$

$$\mathcal{S}_2, \mathcal{S}_1 \text{ set } R^k_{(2,0)} \coloneqq \mathrm{H}_2\big(||_{p\in\mathsf{HH}^k}\,(p, h^p_{(2,0)}, \pi^{p\|0}_{(2,0)}, \pi^{p\|1}_{(2,0)})\big),$$

$$\mathcal{S}_2, \mathcal{S}_0 \text{ set } R^k_{(2,1)} \coloneqq \mathrm{H}_2\big(||_{p\in\mathsf{HH}^k}\,(p, h^p_{(2,1)}, \pi^{p\|0}_{(2,1)}, \pi^{p\|1}_{(2,1)})\big)$$

3. **Batch-Verification.** The servers batch-verify the client inputs for all sessions by invoking π_{check} (Fig. 7.15):

(a) $\mathcal{S}_0$ sets $u_i \coloneqq \{(R^k_{(0,1)}, R^k_{(0,2)}, R^k_{(2,1)}, \widehat{h^{p\|0}}, \widehat{h^{p\|1}})$ for client $i \in [\ell]\}$. $\mathcal{S}_1$ sets $v_i \coloneqq \{(R^k_{(1,0)}, R^k_{(2,0)}, R^k_{(1,2)}, \overline{h^{p\|0}}, \overline{h^{p\|1}})$ for client $i \in [\ell]\}$. $\mathcal{S}_0$ sets $\mathbf{u} \coloneqq \{u_i\}_{i\in[\ell]}$ and $\mathcal{S}_1$ set $\mathbf{v} \coloneqq \{v_i\}_{i\in[\ell]}$. $\mathcal{S}_0$ and $\mathcal{S}_1$ batch-verify all client inputs by computing ver and list L (comprising of invalid client inputs) by running π_{check} with inputs $\mathbf{u}$ and $\mathbf{v}$, respectively as $(\mathsf{ver}, \mathsf{L}) \coloneqq \pi_{\mathsf{check}}(\mathbf{u}, \mathbf{v})$.

$\mathsf{ver} \coloneqq 0$ if $\exists$ a client whose $(R^k_{(0,1)} \neq R^k_{(1,0)}) \vee (R^k_{(0,2)} \neq R^k_{(2,0)}) \vee (R^k_{(2,1)} \neq R^k_{(1,2)}) \vee (\widehat{h^{p\|0}} \neq \overline{h^{p\|0}}) \vee (\widehat{h^{p\|1}} \neq \overline{h^{p\|1}})$, and $\mathsf{L} \coloneqq \{$list of invalid clients' since they failed to pass the above check$\}$. If $\mathsf{ver} = 1$, then all the clients' inputs are valid.

(b) $\mathcal{S}_2$ possesses $R^k_{(2,0)}, R^k_{(2,1)}$ values for each client. $\mathcal{S}_2$ verifies that $\mathcal{S}_2$'s version of $R^k_{(2,1)}$ matches with $\mathcal{S}_0$'s version of $R^k_{(2,1)}$. $\mathcal{S}_2$ also attests that $\mathcal{S}_2$'s version of $R^k_{(2,0)}$ matches with $\mathcal{S}_0$'s version of $R^k_{(0,2)}$ by computing $(\mathsf{ver}', \mathsf{L}')$ as follows:

$$(\mathsf{ver}', \mathsf{L}') \coloneqq \pi_{\mathsf{check}}(\{R^k_{(2,1)}, R^k_{(2,0)}\}_{\ell \text{ clients}} \text{ of } \mathcal{S}_2, \{R^k_{(2,1)}, R^k_{(0,2)}\}_{\ell \text{ clients}} \text{ of } \mathcal{S}_0).$$

(c) $\mathcal{S}_2$ verifies that $\mathcal{S}_2$'s version of $R^k_{(2,0)}$ matches with $\mathcal{S}_1$'s version of $R^k_{(2,0)}$. $\mathcal{S}_2$ also attests that $\mathcal{S}_2$'s version of $R^k_{(2,1)}$ matches with $\mathcal{S}_1$'s version of $R^k_{(1,2)}$ by computing $(\mathsf{ver}'', \mathsf{L}'')$ as follows:

$$(\mathsf{ver}'', \mathsf{L}'') \coloneqq \pi_{\mathsf{check}}(\{R^k_{(2,0)}, R^k_{(2,1)}\}_{\ell \text{ clients}} \text{ of } \mathcal{S}_2, \{R^k_{(2,0)}, R^k_{(1,2)}\}_{\ell \text{ clients}} \text{ of } \mathcal{S}_0).$$

After batch verification, the servers identify the list of bad clients as $\mathsf{L} \coloneqq \mathsf{L} \cup \mathsf{L}' \cup \mathsf{L}''$. The servers ignore the inputs of all clients in L.

Figure 7.8: **Private $\mathcal{T}$-Heavy Hitters Protocol π_{HH}** (continues in Fig. 7.9).

4. **Aggregation.** Aggregate the VIDPF outputs for prefixes $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ as follows: (**Repeated for all validated clients in** $[\ell] \setminus \mathsf{L}$) ▷ (Continued from Fig. 7.8)

$$\mathcal{S}_0 \text{ sets } \mathsf{cnt}^{\gamma}_{(0,1)} \coloneqq \mathsf{cnt}^{\gamma}_{(0,1)} + y^{\gamma}_{(0,1)}, \mathsf{cnt}^{\gamma}_{(0,2)} \coloneqq \mathsf{cnt}^{\gamma}_{(0,2)} + y^{\gamma}_{(0,2)}, \text{ and } \mathsf{cnt}^{\gamma}_{(2,1)} \coloneqq \mathsf{cnt}^{\gamma}_{(2,1)} + y^{\gamma}_{(2,1)}$$

$$\mathcal{S}_1 \text{ sets } \mathsf{cnt}^{\gamma}_{(1,2)} \coloneqq \mathsf{cnt}^{\gamma}_{(1,2)} + y^{\gamma}_{(1,2)}, \mathsf{cnt}^{\gamma}_{(1,0)} \coloneqq \mathsf{cnt}^{\gamma}_{(1,0)} + y^{\gamma}_{(1,0)}, \text{ and } \mathsf{cnt}^{\gamma}_{(2,0)} \coloneqq \mathsf{cnt}^{\gamma}_{(2,0)} + y^{\gamma}_{(2,0)}$$

$$\mathcal{S}_2 \text{ sets } \mathsf{cnt}^{\gamma}_{(2,0)} \coloneqq \mathsf{cnt}^{\gamma}_{(2,0)} + y^{\gamma}_{(2,0)} \text{ and } \mathsf{cnt}^{\gamma}_{(2,1)} \coloneqq \mathsf{cnt}^{\gamma}_{(2,1)} + y^{\gamma}_{(2,1)}$$

The servers have aggregated the VIDPF evaluations (over all the ℓ clients) for all candidate $(k+1)$-bit strings.

5. **Pruning.** Prune the non-heavy hitter strings. For every $(k+1)$-bit string γ, the servers perform the following:

The servers invoke $\mathcal{F}_{\mathsf{CMP}}$ functionality (Fig. 7.10) with the additive shares of the node frequency.

$$\mathcal{S}_0 \text{ invokes } \mathcal{F}_{\mathsf{CMP}}(\mathsf{cnt}^{\gamma}_{(0,1)}, 0, \mathsf{cnt}^{\gamma}_{(0,2)}, \mathsf{cnt}^{\gamma}_{(2,1)}, \mathsf{cnt}^{\gamma}_{(0,2)}, \mathcal{T}),$$

$$\mathcal{S}_1 \text{ invokes } \mathcal{F}_{\mathsf{CMP}}(\mathsf{cnt}^{\gamma}_{(1,0)}, \mathsf{cnt}^{\gamma}_{(1,2)}, 0, \mathsf{cnt}^{\gamma}_{(1,2)}, \mathsf{cnt}^{\gamma}_{(2,0)}, \mathcal{T}),$$

$$\mathcal{S}_2 \text{ invokes } \mathcal{F}_{\mathsf{CMP}}(0, \mathsf{cnt}^{\gamma}_{(2,1)}, \mathsf{cnt}^{\gamma}_{(2,0)}, 0, 0, \mathcal{T})$$

The servers abort if $\mathcal{F}_{\mathsf{CMP}}$ aborts. If $\mathcal{F}_{\mathsf{CMP}}$ outputs 1 set $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \gamma$. Otherwise, the servers ignore γ since it is a non-heavy hitter.

Servers have successfully computed the HH^{k+1} set. Servers repeat *"Server Computation"* steps on $k+1$ bit prefixes.

Output Phase.

The servers output $\mathsf{HH}^{\leq n}$ as the set of $\mathcal{T}$-heavy hitter strings.

Figure 7.9: **Private $\mathcal{T}$-Heavy Hitters Protocol π_{HH}** (continuing from Fig. 7.8).

Server Computation. Each server first initializes a set of sets for heavy-hitter computation as $\mathsf{HH}^{\leq n} := \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\} := \{\epsilon, \emptyset, \ldots, \emptyset\}$, where HH^0 contains empty string ϵ, $\mathsf{HH}^1, \ldots, \mathsf{HH}^n$ are empty sets and HH^k corresponds to the kth level. The servers start accepting VIDPF keys from the clients. As in our histogram protocol, $\mathcal{S}_2$ acts as an attesting server for the sessions involving keys $\mathsf{key}_{(2,0)}$ and $\mathsf{key}_{(2,1)}$ by sending hashes (depicted in Fig. 7.2). Next, for $k \in [n]$ the servers perform the following:

Initialization. For each k-bit heavy-hitting prefix $p \in \mathsf{HH}^k$, the servers initialize to 0 a $\mathsf{cnt}^{p\|0}$ (resp. $\mathsf{cnt}^{p\|1}$) variable for each session to count the frequency of prefix $p \,\|\, 0$ (resp. $p \,\|\, 1$). Each server aggregates for each of the three sessions their additive shares of each frequency in their local cnt variables and uses them for pruning.

VIDPF Evaluation. Next, the servers retrieve from memory the states for VIDPF evaluation in all three sessions corresponding to prefix $p \in \{0,1\}^k$ for each client. These states are used to incrementally evaluate the VIDPF on prefix strings $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ for every client in all three sessions. For each client, the servers obtain new evaluation states (corresponding to prefix γ), VIDPF output for prefix string γ, and proof strings. The states are stored in memory for future VIDPF evaluations on $\gamma \,\|\, 0$ and $\gamma \,\|\, 1$ in the $(k+1)^{th}$ level. More formally, the servers compute a secret shared vector $y^{\gamma}_{(b_1,b_2)}$ and a hash $\pi^{\gamma}_{(b_1,b_2)}$ that is used for consistency checking by relying on the verifiability property of the VIDPF. Next, the servers validate the client's input. If $k = 1$, then the servers reconstruct $y^0 + y^1$ for each client to verify that $y^0 + y^1 = 1$. If $k \neq 1$, then the servers reconstruct $y^p - (y^{p\|0} + y^{p\|1})$ and verify that it is 0. This ensures that the subtrees involving $p \,\|\, 0$ and $p \,\|\, 1$ are valid. The servers also need to ensure that the client has provided a consistent input across the three sessions. This is ensured by computing the difference of the reconstructed outputs across the sessions and verifying that they equal 0 by matching their hash values with the other servers' hash in Step 2b(v) of Fig. 7.7.

Batch-Verification. The servers need to check: (1) that the hashes they possess for a client are equal, and (2) that $y^p = (y^{p\|0} + y^{p\|1})$. Both these checks are reduced to checking the equality of a string (corresponding to each client) held by servers.

Let $\mathbf{u}$ (resp. $\mathbf{v}$) be the list of ℓ (one for each client) strings held by the first (resp. second) server. Then, the servers perform a batch verification of $\mathbf{u}$ and $\mathbf{v}$ strings by invoking the subprotocol $\pi_{\mathsf{check}}(\mathbf{u}, \mathbf{v})$ in Fig. 7.15. If the two lists $\mathbf{u}$ and $\mathbf{v}$ are equal then π_{check} returns $\mathsf{ver} = 1$, else it returns $\mathsf{ver} = 0$ and a list L containing the indices of elements where the lists differ. This is performed for all three sessions. $\mathcal{S}_2$ also attests the sessions that it is involved in. This is performed using batch-verification, yielding output lists L' and L''. Finally, the servers identify the list of bad clients as $\mathsf{L} = \mathsf{L} \cup \mathsf{L}' \cup \mathsf{L}''$ and their VIDPF output is ignored. The servers consider the rest of the clients as "validated" and they are moved to the aggregation phase.

Aggregation. Once a client's VIDPF output y^γ is validated for $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$, it is aggregated into $\mathsf{cnt}^\gamma \coloneqq \mathsf{cnt}^\gamma + y^\gamma$. This is locally performed by each server (for all three sessions) using the secret shares of y^γ since it only involves addition. The servers perform this over every validated client output, and at the end of this phase, the servers possess a secret share of the frequency of $p \,\|\, 0$ and $p \,\|\, 1$ as $\mathsf{cnt}^{p\|0}$ and $\mathsf{cnt}^{p\|1}$.

Pruning. The servers proceed to pruning and invoke $\mathcal{F}_{\mathsf{CMP}}$ (Fig. 7.10) on the secret shares of cnt^γ (for $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$) for all sessions and threshold $\mathcal{T}$. Based on the output of $\mathcal{F}_{\mathsf{CMP}}$ the following occurs:

- $\mathcal{F}_{\mathsf{CMP}}$ returns 1 if $\mathsf{cnt}^\gamma \geq \mathcal{T}$ (i.e., γ is a heavy-hitter string). In this case, the prefix γ is added to the list of $k+1$-bit heavy-hitter set (i.e., $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \gamma$).

- $\mathcal{F}_{\mathsf{CMP}}$ returns 0 if $\mathsf{cnt}^\gamma < \mathcal{T}$ (i.e., γ is a non heavy-hitter string). In this case, the prefix γ is ignored.

- If $\mathcal{F}_{\mathsf{CMP}}$ returns $\perp$, then one of the servers behaved maliciously and the honest servers abort. This occurs if the malicious server has provided an incorrect threshold as input (condition 1 in $\mathcal{F}_{\mathsf{CMP}}$) or it provided incorrect client output shares as input (condition 4 in $\mathcal{F}_{\mathsf{CMP}}$).

This computation is performed in parallel for all $(k+1)$-bit prefixes in consideration, and after the pruning phase, HH^{k+1} contains the list of $(k+1)$-bit heavy hitter strings.

Next, the above computation is repeated for $(k+1)$-bit strings to compute $(k+2)$-bit heavy hitters, until we reach $k = n-1$. As already mentioned, $\mathcal{F}_{\mathsf{CMP}}$ is securely implemented using the state-of-the-art protocol of Rabbit [171].

Functionality $\mathcal{F}_{\mathsf{CMP}}$

Inputs: Party P_0 has input $(a_0, b_0, c_0, d_0, e_0, \mathcal{T}_0)$, Party P_1 has input $(a_1, b_1, c_1, d_1, e_1, \mathcal{T}_1)$, and Party P_2 has input $(a_2, b_2, c_2, d_2, e_2, \mathcal{T}_2)$.

Outputs: Compute $a \coloneqq a_0 + a_1, b \coloneqq b_1 + b_2, c \coloneqq c_0 + c_2$, $d \coloneqq d_0 + d_1, e \coloneqq e_1 + e_2$, and proceed as follows:

1. If $\mathcal{T}_0 \neq \mathcal{T}_1 \neq \mathcal{T}_2$, then $\mathcal{F}_{\mathsf{CMP}}$ aborts. Else, set $\mathcal{T} \coloneqq \mathcal{T}_0$.
2. If $a = b = c = d = e$ and $a < \mathcal{T}$ output 0.
3. If $a = b = c = d = e$ and $a \geq \mathcal{T}$ output 1.
4. Else, $\mathcal{F}_{\mathsf{CMP}}$ aborts (i.e. a, b, c, d or e strings are not equal).

Corruption: Adversary $\mathcal{A}$ maliciously corrupts one server. If $\mathcal{A}$ instructs the functionality to abort by sending $\perp$, the functionality instructs the honest servers to abort.

Figure 7.10: The ideal $\mathcal{F}_{\mathsf{CMP}}$ functionality for comparison.

Output Phase. At the end, the servers output $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots, \mathsf{HH}^n\}$ as the set of $\mathcal{T}$-heavy hitter strings.

This completes the description of π_{HH} (Figs. 7.7, 7.8). Security of our protocol is captured in Theorem 4 and proven next.

7.4 Proof of Heavy-Hitters Protocol π_{HH}

Theorem 4. *Assuming VIDPF is a verifiable incremental DPF and* $\mathrm{H}_1, \mathrm{H}_2$ *are random oracles,* $\mathcal{F}_{\mathsf{CMP}}$ *is a secure comparison functionality (Fig. 7.10), and* H *(in* π_{check}*) is collision-resistant, then* π_{HH} *(Figs. 7.7 and 7.8) implements* $\mathcal{F}_{\mathsf{HH}}$ *in the (random oracle,* $\mathcal{F}_{\mathsf{CMP}}$*)-model against malicious corruption of one server and* $\ell' \leq \ell$ *clients.*

7.4.1 Proof Sketch

Proof. The adversary is allowed to corrupt $\ell' \leq \ell$ clients and one of the servers. The other two servers are honest. We discuss the ways a malicious client can attempt to inject an error and we demonstrate our consistency checks for them:

- *Client VIDPF keys are malformed.* A malicious client can attempt to provide malformed VIDPF keys which are non-zero in more than one path in the binary tree (of 2^n leaves). This gets detected in the session involving the honest servers due to the verifiable property of the VIDPF at each level when the servers verify the proofs generated during the VIDPF evaluation. If the checks pass, then it is ensured that the VIDPF keys provided by the client are valid.

- *Client VIDPF input is malformed.* Next, a malicious client can try to double-vote on an input point, say $p \,\|\, 0 \in \{0,1\}^{k+1}$ by constructing the VIDPF on $(p \,\|\, 0, \widetilde{\beta^k})$, i.e., $f(p \,\|\, 0) = \widetilde{\beta^k}$, where $\widetilde{\beta^k} > 1$, instead of $(p \,\|\, 0, 1)$. This is detected by the honest servers since they perform a local subtree verification by reconstructing the value $y^p - (y^{p\|0} - y^{p\|1})$ and verifying that it equals 0 for all $k > 0$. For $k = 0$, the servers verify that $y^\epsilon = 1$. Combining all k checks ensures that $y^{p\|0} = 1$ if and only if $y^p = 1$ and $y^{p\|1} = 0$, else $y^{p\|0} = 0$.

- *VIDPF input is inconsistent across sessions.* Finally, a malicious client can try to provide different VIDPF keys in different sessions. For example it constructs VIDPF keys for input $(\alpha_1, 1)$ for the $\mathcal{S}_0 - \mathcal{S}_1$ session and $(\alpha_2, 1)$ for the $\mathcal{S}_1 - \mathcal{S}_2$ session and $(\alpha_3, 1)$ for the $\mathcal{S}_2 - \mathcal{S}_0$ session, where $\alpha_1 \neq \alpha_2 \neq \alpha_3$ and $\alpha_1, \alpha_2, \alpha_3 \in \{0,1\}^k$. The above two checks would still pass since they ensure client input validation within each session but not client input consistency across the sessions. To ensure this, the servers match the difference of the reconstructed output of $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_2 - \mathcal{S}_0$ session, and the difference of the reconstructed output of $\mathcal{S}_2 - \mathcal{S}_0$ and $\mathcal{S}_1 - \mathcal{S}_2$ session, to verify that they are all 0. By transitivity, it is ensured that if and only if this check passes then the output of the VIDPF evaluation would be the same across the three sessions, ensuring that $\alpha_1 = \alpha_2 = \alpha_3$. This is performed by computing the $\widehat{h^{p\|0}}$ and $\widehat{h^{p\|1}}$ hashes for every heavy-hitting prefix p computed by π_{HH}.

A malicious server could collude with malicious clients. It can be observed that the honest clients' inputs are always hidden from the adversary due to input privacy of

VIDPF, since no server possesses more than one VIDPF key. Next, A malicious server could attempt to incorporate an erroneous VIDPF evaluation (from a malformed client input key) or inject additive errors into the output. We show how this is tackled in the protocol based on the server corruption:

- $\mathcal{S}_0$ *is corrupt.* In this case, the session between $\mathcal{S}_1 - \mathcal{S}_2$ is honest. $\mathcal{S}_0$ runs this session with $\mathcal{S}_1$ since it obtained $\mathsf{key}_{(2,1)}$ from the client. However, $\mathcal{S}_2$ behaves as an attestator by sending hashes of the messages that $\mathcal{S}_0$ is supposed to send. This forces $\mathcal{S}_0$ to act honestly in the $\mathcal{S}_1 - \mathcal{S}_2$, otherwise, it leads to an abort. Another way a malicious $\mathcal{S}_0$ can behave badly is by colluding with a malicious client. The client could provide malformed inputs in $\mathcal{S}_0 - \mathcal{S}_1/\mathcal{S}_2 - \mathcal{S}_0$ session or inconsistent inputs across the three sessions. In such a case, a malicious $\mathcal{S}_0$ could compute an incorrect hash $\widehat{h^{p\|0}} \coloneqq \mathrm{H}_1(y_{(0,1)}^{p\|0\,\prime} - y_{(0,2)}^{p\|0\,\prime}, y_{(0,2)}^{p\|0\,\prime} - y_{(2,1)}^{p\|0})$ and $\widehat{h^{p\|1}} \coloneqq \mathrm{H}_1(y_{(0,1)}^{p\|1\,\prime} - y_{(0,2)}^{p\|1\,\prime}, y_{(0,2)}^{p\|1\,\prime} - y_{(2,1)}^{p\|1})$ where $y_{(0,1)}^{p\|0\,\prime}, y_{(0,2)}^{p\|0\,\prime}, y_{(0,1)}^{p\|1\,\prime}, y_{(0,2)}^{p\|1\,\prime}$ are incorrect. This would allow $\mathcal{S}_0$ to introduce an additive error into the frequency for $p \,\|\, 0$ and $p \,\|\, 1$ (for the $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_2 - \mathcal{S}_0$ sessions) by incorporating the client's malformed input. However, this gets detected when the output count is secretly reconstructed by the $\mathcal{F}_{\mathsf{CMP}}$ functionality for all three sessions and compared. The reconstructed count won't match and the ideal functionality would return a $\perp$ message detecting that one of the servers behaved maliciously, leading to an abort in the π_{HH}.

- $\mathcal{S}_1$ *is corrupt.* This case is very similar to the above one where $\mathcal{S}_0$ was corrupt. In this case, the session between $\mathcal{S}_2 - \mathcal{S}_0$ is honest. $\mathcal{S}_1$ runs this session with $\mathcal{S}_0$ since it obtained $\mathsf{key}_{(2,0)}$ from the client. However, $\mathcal{S}_2$ behaves as an attestator by sending hashes of the messages that $\mathcal{S}_1$ is supposed to send. This forces $\mathcal{S}_1$ to act honestly in the $\mathcal{S}_2 - \mathcal{S}_0$, otherwise, it leads to an abort. Another way a malicious $\mathcal{S}_1$ can behave badly is by colluding with a malicious client. The client could provide malformed inputs in $\mathcal{S}_0 - \mathcal{S}_1/\mathcal{S}_1 - \mathcal{S}_2$ session or inconsistent inputs across the three sessions. In such a case, a malicious $\mathcal{S}_1$ simply ignores the hash values $\widehat{h^{p\|0}}$ and $\widehat{h^{p\|1}}$ sent by $\mathcal{S}_0$. This would allow the $\mathcal{S}_1$ to introduce an additive

error into the frequency for $p \| 0$ and $p \| 1$ (for the $\mathcal{S}_0 - \mathcal{S}_1$ and $\mathcal{S}_1 - \mathcal{S}_2$ sessions) by incorporating the client's malformed input. However, this gets detected when the output count is secretly reconstructed by the $\mathcal{F}_{\mathsf{CMP}}$ functionality for all three sessions and compared. The reconstructed count won't match and the ideal functionality would return a $\perp$ message detecting that one of the servers behaved maliciously, leading to an abort in the π_{HH}.

- $\mathcal{S}_2$ *is corrupt.* In this case, the session between $\mathcal{S}_0 - \mathcal{S}_1$ is honest. If $\mathcal{S}_2$ behaves as a malicious attestator by sending incorrect hashes for the $\mathcal{S}_1 - \mathcal{S}_2$ or $\mathcal{S}_2 - \mathcal{S}_0$ sessions then the honest servers abort. Another way a malicious $\mathcal{S}_2$ can behave badly is by colluding with a malicious client. The client could provide malformed inputs in the three sessions. If the client provides malformed inputs in $\mathcal{S}_0 - \mathcal{S}_1$ session then it gets detected due to verifiability of the VIDPF and the local subtree verification, since both $\mathcal{S}_0$ and $\mathcal{S}_1$ are honest. It could provide malformed (allows double voting) VIDPF keys $\mathsf{key}'_{(2,0)}$ and $\mathsf{key}'_{(2,1)}$ to $\mathcal{S}_1$ and $\mathcal{S}_0$ for the sessions involving $\mathcal{S}_2$. However, that again gets detected since the server $\mathcal{S}_0$ computes the hashes $\widehat{h^{p\|0}}$ and $\widehat{h^{p\|1}}$ honestly and the $\mathcal{S}_1$ verifies them honestly.

□

7.4.2 Formal Proof Details of Theorem 4

Security of our protocol relies on the correctness of π_{check}. π_{check} is a protocol where two honest parties commit to their inputs using Merkle-tree-based commitments and then they decommit based on whether the root commitments match or not. Correctness of π_{check} follows in a straightforward manner from the binding property of the Merkle-tree commitment, which in turn follows from the collision-resistance property of the hash function used in π_{check}.

Next, we prove the security of our protocol in the real-ideal world paradigm of Canetti [66]. Let $\mathcal{A}$ denote the real-world adversary corrupting one of the servers and ℓ' clients maliciously in the real-world execution of the protocol. Let $\text{REAL}_{\mathcal{A},\pi_{\mathsf{HH}}}$

denote $\mathcal{A}$'s view after participating in the real-world execution. Let simulator Sim be the ideal-world adversary, which given access to the algorithm of $\mathcal{A}$ and functionality $\mathcal{F}_{\mathsf{HH}}$, produces the ideal world adversarial view as $\text{IDEAL}_{\mathsf{Sim},\mathcal{F}_{\mathsf{HH}}}$.

We prove that our protocol π_{HH} securely implements $\mathcal{F}_{\mathsf{HH}}$ functionality by providing an ideal world PPT simulator Sim for all PPT adversaries $\mathcal{A}$, and show that the real and ideal world view are indistinguishable, i.e., $\text{REAL}_{\mathcal{A},\pi_{\mathsf{HH}}} \overset{c}{\approx} \text{IDEAL}_{\mathsf{Sim},\mathcal{F}_{\mathsf{HH}}}$. We use a sequence of hybrids (i.e., HYB_0 - HYB_4) to prove the indistinguishability argument.

Proof. We first consider the case where $\mathcal{A}$ corrupts server $\mathcal{S}_2$ along with ℓ' clients. Then, we consider the case where $\mathcal{A}$ corrupts either $\mathcal{S}_0$ or $\mathcal{S}_1$ along with ℓ' clients.

$\mathcal{S}_2$ *is corrupt.* We provide the formal simulator in Fig. 7.11 and argue indistinguishability as follows.

HYB_0 : The real world execution of the protocol.

HYB_1 : Same as HYB_0, except Sim aborts if a malicious client i has provided inconsistent u_i and v_i inputs to $\mathcal{S}_0$ and $\mathcal{S}_1$ and yet passed the batched consistency check π_{check}. The two hybrids are indistinguishable due to the correctness of π_{check}.

HYB_2 : Same as HYB_1, except the Sim extracts the corrupt client's inputs using the three pairs of DPF keys. Then Sim runs Step c of simulated Batch-Verification, i.e., Sim aborts if 1) the client's input α_i is k-bits heavy-hitting, 2) $\alpha_i \,\|\, 0$ or $\alpha_1 \,\|\, 1$ is invalid, and 3) client evaded the Batch-Verification check for the sessions run between honest servers. The two hybrids are indistinguishable due to the verifiability property of VIDPF in the random oracle model. This occurs when the client successfully evades the input extraction process of VIDPF by providing malformed VIDPF keys and yet passes the batch verification checks.

HYB_3 : Same as HYB_2, except Sim invokes $\mathcal{F}_{\mathsf{HH}}$ with the extracted inputs to obtain the $\mathsf{HH}^{\leq n}$ set and simulates $\mathcal{F}_{\mathsf{CMP}}$ based on whether a prefix γ is in $\mathsf{HH}^{\leq n}$ or not. The two hybrids are indistinguishable against a corrupt server $\mathcal{S}_2$ in the $\mathcal{F}_{\mathsf{CMP}}$-model.

Simulator Sim for maliciously corrupt ℓ' number of clients and server $\mathcal{S}_2$

- **Corruption:** Server $\mathcal{S}_2$ and ℓ' number of clients are maliciously corrupt. The rest $\ell - \ell'$ clients and servers $(\mathcal{S}_0, \mathcal{S}_1)$ are simulated by simulator Sim.
- **Primitive:** VIDPF $:= (\mathsf{Gen}, \mathsf{EvalPref}, \mathsf{EvalNext})$ is a verifiable incremental DPF. $\mathrm{H}_1, \mathrm{H}_2 : \{0,1\}^* \to \{0,1\}^\kappa$ are random oracles.

Client $\mathcal{C}$ Computation. (**Repeated for ℓ clients**)

1. *If the client is honest:* Sim simulates the client by preparing three pairs of DPF keys with input 1 and output values $(1, \ldots, 1)$ as follows: $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(1,0)}) := \mathsf{Gen}(1^\kappa, 1^n, 1, (1, \ldots, 1), \mathbb{G})$, $(\mathsf{key}_{(1,2)}, \mathsf{key}_{(2,1)}) := \mathsf{Gen}(1^\kappa, 1^n, 1, (1, \ldots, 1), \mathbb{G})$, $(\mathsf{key}_{(2,0)}, \mathsf{key}_{(0,2)}) := \mathsf{Gen}(1^\kappa, 1^n, 1, (1, \ldots, 1), \mathbb{G})$.

   Sim sends $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(0,2)}, \mathsf{key}_{(2,1)})$ to $\mathcal{S}_0$, $(\mathsf{key}_{(1,0)}, \mathsf{key}_{(1,2)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_1$ and $(\mathsf{key}_{(2,1)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_2$ on behalf of the client.

2. *If the client is corrupt:* Client sends $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(0,2)}, \mathsf{key}_{(2,1)})$ to $\mathcal{S}_0$, $(\mathsf{key}_{(1,0)}, \mathsf{key}_{(1,2)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_1$ and $(\mathsf{key}_{(2,1)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_2$.

Server Computation.
(Simulator Sim initializes a list $\mathsf{L}_{ext} = \{\}$ and $\mathsf{L}_{inp} = \{\}$, and simulates $\mathcal{S}_0$ and $\mathcal{S}_1$)
For each corrupt client i, the simulator performs the following for input extraction: (**Repeated for ℓ' corrupt clients**)

1. Sim extracts the corrupt client's input $(\alpha'_i, \beta'_{i,1}, \ldots, \beta'_{i,n})$ from the three pairs of DPF keys - $\mathsf{key}_{(0,1)}$ and $\mathsf{key}_{(1,0)}$, $\mathsf{key}_{(0,2)}$ and $\mathsf{key}_{(2,0)}$, and $\mathsf{key}_{(2,1)}$ and $\mathsf{key}_{(1,2)}$, provided by client i. If the extracted values differ, then Sim takes the necessary steps below.
2. If the corrupt client has not provided a valid input at level j, i.e., 1) $\exists j \in [n]$ s.t. $\beta'_j \neq 1$ (for the smallest j), or 2) the extracted inputs α'_i (from the three sessions) in the previous step differ in the j^{th} bit, i.e., $\alpha'_{i,j}$, then Sim truncates the extracted input of client i to the first j bits of α_i as $\alpha_i := \alpha_{i,\leq j-1}$. Sim sets $\mathsf{L}^{j-1}_{\mathsf{ext}} = \mathsf{L}^{j-1}_{\mathsf{ext}} \cup \{i, j-1\}$ and updates $\mathsf{L}_{\mathsf{ext}} = \mathsf{L}_{\mathsf{ext}} \cup \mathsf{L}^{j-1}_{\mathsf{ext}}$ to denote that the ith client's input is valid only till level $j-1$.
3. Sim stores the extracted input (after necessary truncation) α_i for client i in a list $\mathsf{L}_{\mathsf{inp}}$ as $\mathsf{L}_{\mathsf{inp}} := \mathsf{L}_{\mathsf{inp}} \cup \{i, \alpha_i\}$.

After running the above extraction process for all corrupt clients, Sim invokes $\mathcal{F}_{\mathsf{HH}}$ with the input list $\mathsf{L}_{\mathsf{inp}}$ to obtain the output set of $\mathcal{T}$-heavy hitting prefixes as $\mathsf{HH}^{\leq n}$. The functionality $\mathcal{F}_{\mathsf{HH}}$ waits for further instructions from the ideal world adversary Sim.

For $k \in [0, \ldots, n-1]$ repeat the following steps: ▷ n is the number of bits.

1. **Initialization.** For prefix $p \in \mathsf{HH}^k$, Sim initialize server $\mathcal{S}_0$'s and $\mathcal{S}_1$'s aggregation variables for prefixes $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ as follows: Simulated $\mathcal{S}_0$ sets $\mathsf{cnt}^\gamma_{(0,1)} := \mathsf{cnt}^\gamma_{(0,2)} := \mathsf{cnt}^\gamma_{(2,1)} := 0$ and Simulated $\mathcal{S}_1$ sets $\mathsf{cnt}^\gamma_{(1,2)} := \mathsf{cnt}^\gamma_{(1,0)} := \mathsf{cnt}^\gamma_{(2,0)} := 0$.
2. **VIDPF Evaluation.** For prefix $p \in \mathsf{HH}^{\leq k}$, Sim simulates $\mathcal{S}_0$ and $\mathcal{S}_1$ by running the original protocol steps. (**Repeated for ℓ clients**)
3. **Batch-Verification.**
   (a) Sim simulates $\mathcal{S}_0$ and $\mathcal{S}_1$ by computing $\mathbf{u}$ and $\mathbf{v}$ following the original steps of the protocol and Sim adds the ith client to the list L of discarded clients if $u_i \neq v_i$. If client i is not detected as bad by running the original protocol steps of π_{check} on $\mathbf{u}$ and $\mathbf{v}$ then Sim aborts.
   (b) Sim runs the honest protocol steps to simulate the interaction between $\mathcal{S}_2 - \mathcal{S}_0$ and $\mathcal{S}_2 - \mathcal{S}_1$ to obtain the update list L.
   (c) Sim aborts if $\exists$ client i s.t. 1) its input is k-bits heavy-hitting (i.e., $\alpha_i \in \mathsf{HH}^k$), 2) $\alpha_i \,\|\, 0$ or $\alpha_i \,\|\, 1$ is not valid, i.e., $\{i, k\} \in \mathsf{L}^k_{\mathsf{ext}}$, 3) client i evaded the consistency check, i.e., $i \notin \mathsf{L}$.

   If Sim did not abort then for all corrupt parties in list L at level k, Sim invokes $\mathcal{F}_{\mathsf{HH}}$ to discard the parties from the output computation of $k+1$-bit heavy-hitting prefixes. Sim obtains an updated $\mathsf{HH}^{\leq n}$ set from $\mathcal{F}_{\mathsf{HH}}$.

Figure 7.11: Simulation Algorithm against malicious corruption of server $\mathcal{S}_2$ and ℓ' clients. Continues in Fig. 7.12.

4. **Aggregation.** Sim simulates this step for prefixes $\gamma \in \{p \| 0, p \| 1\}$ as follows: **(Repeated for all validated clients in $[\ell] \setminus \mathsf{L}$)**

   Simulated $\mathcal{S}_0$ sets $\mathsf{cnt}^{\gamma}_{(0,1)} \coloneqq \mathsf{cnt}^{\gamma}_{(0,2)} \coloneqq \mathsf{cnt}^{\gamma}_{(2,1)} \coloneqq 0$, Simulated $\mathcal{S}_1$ sets $\mathsf{cnt}^{\gamma}_{(1,2)} \coloneqq \mathsf{cnt}^{\gamma}_{(1,0)} \coloneqq \mathsf{cnt}^{\gamma}_{(2,0)} \coloneqq 0$.

5. **Pruning.** For every $(k+1)$-bit string γ, Sim simulates the pruning step as follows:

   If $\gamma \in \mathsf{HH}^{k+1}$ **then** Sim invokes the simulator of $\mathcal{F}_{\mathsf{CMP}}$ with output 1 s.t. $\mathcal{F}_{\mathsf{CMP}}$ returns 1 as output to the servers, s.t. γ is included in the list of heavy-hitting strings.

   If $\gamma \notin \mathsf{HH}^{k+1}$ **then** Sim invokes the simulator of $\mathcal{F}_{\mathsf{CMP}}$ with output 0 s.t. $\mathcal{F}_{\mathsf{CMP}}$ returns 0 as output to the servers, s.t. γ gets pruned.

   If the simulator of $\mathcal{F}_{\mathsf{CMP}}$ aborts, then Sim instructs $\mathcal{F}_{\mathsf{HH}}$ to abort at level $(\perp, k+1)$ and Sim aborts this simulated execution. Sim has successfully simulated the HH^{k+1} set. Sim repeats *"Server Computation"* steps (starting from Step 2b) on $k+1$ bit prefixes.

Output Phase.

Sim outputs $\mathsf{HH}^{\leq n}$ as the set of $\mathcal{T}$-heavy hitter strings on behalf of simulated $\mathcal{S}_0$ and $\mathcal{S}_1$, and instructs $\mathcal{F}_{\mathsf{HH}}$ to send output to the honest servers $\mathcal{S}_0$ and $\mathcal{S}_1$.

Figure 7.12: Continuing the simulation from Fig. 7.11. Algorithm against malicious corruption of server $\mathcal{S}_2$ and ℓ' clients.

HYB_4 : Same as HYB_3, except Sim simulates the DPF key generation for the honest clients with input $(\alpha, (\beta_1, \ldots, \beta_n)) = (1, (1, \ldots, 1))$ and sets the counters to 0s in the aggregation step. Indistinguishable due to VIDPF input privacy. The 0-valued counters in the aggregation step are identically distributed to the actual aggregation counters since HYB_3 and HYB_4 are in the $\mathcal{F}_{\mathsf{CMP}}$-model. This is the ideal world execution of the protocol, completing our simulation algorithm.

Either $\mathcal{S}_0$ or $\mathcal{S}_1$ is corrupt. Next, we consider the case where either server $\mathcal{S}_0$ or $\mathcal{S}_1$ is corrupted along with ℓ' clients. We provide the simulator in Fig. 7.13 and argue indistinguishability as follows. (This case is similar to the case where $\mathcal{S}_1$ is corrupted along with ℓ' clients.)

HYB_0 : The real world execution of the protocol.

HYB_1 : Same as HYB_0, except Sim aborts if a malicious client i has provided values $(R^k_{(2,0)}, R^k_{(2,1)})$ to $\mathcal{S}_2$ and values $(R^k_{(2,0)}, R^k_{(1,2)})$ to $\mathcal{S}_1$ such that they are not equal, and yet client i passed the batched consistency check π_{check}. The two hybrids are indistinguishable due to the correctness of π_{check}.

Simulator Sim for maliciously corrupt ℓ' number of clients and server $\mathcal{S}_0$

- **Corruption:** ℓ' number of clients and server $\mathcal{S}_0$ are maliciously corrupt. The rest $\ell - \ell'$ clients and servers $(\mathcal{S}_1, \mathcal{S}_2)$ are simulated by simulator Sim. Without loss of generality, we will assume that $\mathcal{S}_0$ is corrupt; the case where $\mathcal{S}_1$ is corrupt is symmetric.
- **Primitive:** VIDPF $\coloneqq (\mathsf{Gen}, \mathsf{EvalPref}, \mathsf{EvalNext})$ is a verifiable incremental DPF. $\mathrm{H}_1, \mathrm{H}_2 : \{0,1\}^* \to \{0,1\}^\kappa$ are random oracles.

Client $\mathcal{C}$ Computation. (**Repeated for ℓ clients**)

1. *If the client is honest:* Sim simulates the client by preparing three pairs of DPF keys with input 1 and output values $(1, \ldots, 1)$. $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(1,0)}) \coloneqq \mathsf{Gen}(1^\kappa, 1^n, 1, (1, \ldots, 1), \mathbb{G})$, $(\mathsf{key}_{(1,2)}, \mathsf{key}_{(2,1)}) \coloneqq \mathsf{Gen}(1^\kappa, 1^n, 1, (1, \ldots, 1), \mathbb{G})$, $(\mathsf{key}_{(2,0)}, \mathsf{key}_{(0,2)}) \coloneqq \mathsf{Gen}(1^\kappa, 1^n, 1, (1, \ldots, 1), \mathbb{G})$

   Sim sends $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(0,2)}, \mathsf{key}_{(2,1)})$ to $\mathcal{S}_0$, $(\mathsf{key}_{(1,0)}, \mathsf{key}_{(1,2)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_1$ and $(\mathsf{key}_{(2,1)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_2$ on behalf of the client.

2. *If the client is corrupt:* Client sends $(\mathsf{key}_{(0,1)}, \mathsf{key}_{(0,2)}, \mathsf{key}_{(2,1)})$ to $\mathcal{S}_0$, $(\mathsf{key}_{(1,0)}, \mathsf{key}_{(1,2)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_1$ and $(\mathsf{key}_{(2,1)}, \mathsf{key}_{(2,0)})$ to $\mathcal{S}_2$.

Server Computation.
(Simulator Sim initializes a list $\mathsf{L}_{ext} = \{\}$ and $\mathsf{L}_{inp} = \{\}$, and simulates $\mathcal{S}_1$ and $\mathcal{S}_2$)
For each corrupt client i, the simulator performs the following for input extraction: (**Repeated for ℓ' corrupt clients**)

1. Sim extracts the corrupt client's input $(\alpha'_i, \beta'_{i,1}, \ldots, \beta'_{i,n})$ from the pair of DPF keys - $\mathsf{key}_{(1,2)}$ and $\mathsf{key}_{(2,1)}$, provided by client i.
2. If the corrupt client has not provided a valid input at level j, i.e., $\exists j \in [n]$ s.t. $\beta'_j \neq 1$ (for the smallest j), then Sim truncates the extracted input of client i to the first j bits of α_i as $\alpha_i \coloneqq \alpha_{i,\leq j-1}$. Sim sets $\mathsf{L}_{\mathsf{ext}}^{j-1} = \mathsf{L}_{\mathsf{ext}}^{j-1} \cup \{i, j-1\}$ and updates $\mathsf{L}_{\mathsf{ext}} = \mathsf{L}_{\mathsf{ext}} \cup \mathsf{L}_{\mathsf{ext}}^{j-1}$ to denote that the ith client's input is valid only till level $j-1$.
3. Sim stores the extracted input (after necessary truncation) α_i for client i in a list $\mathsf{L}_{\mathsf{inp}}$ as $\mathsf{L}_{\mathsf{inp}} \coloneqq \mathsf{L}_{\mathsf{inp}} \cup \{i, \alpha_i\}$.

After running the above extraction process for all corrupt clients, Sim invokes $\mathcal{F}_{\mathsf{HH}}$ with the input list $\mathsf{L}_{\mathsf{inp}}$ to obtain the output set of $\mathcal{T}$-heavy hitting prefixes as $\mathsf{HH}^{\leq n}$. The functionality $\mathcal{F}_{\mathsf{HH}}$ waits for further instructions from the ideal world adversary Sim.

For $k \in [0, \ldots, n-1]$ repeat the following steps: ▷ n is the number of bits.

1. **Initialization.** For prefix $p \in \mathsf{HH}^k$, Sim initialize server $\mathcal{S}_1$'s and $\mathcal{S}_2$'s aggregation variables for prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ as follows: Simulated $\mathcal{S}_1$ sets $\mathsf{cnt}^\gamma_{(1,2)} \coloneqq \mathsf{cnt}^\gamma_{(1,0)} \coloneqq \mathsf{cnt}^\gamma_{(2,0)} \coloneqq 0$ and simulated $\mathcal{S}_2$ sets $\mathsf{cnt}^\gamma_{(2,0)} \coloneqq \mathsf{cnt}^\gamma_{(2,1)} \coloneqq 0$.
2. **VIDPF Evaluation.** For prefix $p \in \mathsf{HH}^{\leq k}$, Sim simulates $\mathcal{S}_1$ and $\mathcal{S}_2$ by running the original protocol steps. (**Repeated for ℓ clients**)

Figure 7.13: Simulation Algorithm against malicious corruption of server $\mathcal{S}_0$ and ℓ' clients. Continues in Fig. 7.14.

3. **Batch-Verification.**

   (a) Sim simulates the interaction between corrupt server $\mathcal{S}_0$ and honest server $\mathcal{S}_1$ by following the protocol steps to update list L.

   (b) Sim simulates the interaction between corrupt server $\mathcal{S}_0$ and honest server $\mathcal{S}_2$ by following the protocol steps to update list L.

   (c) For each client i: Sim verifies that $\mathcal{S}_2$'s version of $(R^k_{(2,0)}, R^k_{(2,1)})$ matches with $\mathcal{S}_1$'s version of $(R^k_{(2,0)}, R^k_{(1,2)})$. If they don't match then Sim adds ith client to the list L of discarded clients. If client i is not detected as bad by running the original protocol steps of π_{check} between $\mathcal{S}_1$ and $\mathcal{S}_2$ then Sim aborts.

   (d) Sim aborts if $\exists$ client i s.t. 1) its input is k-bits heavy-hitting (i.e., $\alpha_i \in \mathsf{HH}^k$), 2) $\alpha_i \,\|\, 0$ or $\alpha_i \,\|\, 1$ is not valid, i.e., $\{i, k\} \in \mathsf{L}^k_{\mathsf{ext}}$, 3) client i evaded the consistency check, i.e., $i \notin \mathsf{L}$.

   If Sim did not abort then for all corrupt parties in list L at level k, Sim invokes $\mathcal{F}_{\mathsf{HH}}$ to discard the parties from the output computation of $k+1$-bit heavy-hitting prefixes. Sim obtains an updated $\mathsf{HH}^{\leq n}$ set from $\mathcal{F}_{\mathsf{HH}}$.

4. **Aggregation.** Sim simulates this step for prefixes $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ as follows: **(Repeated for all validated clients in $[\ell] \setminus \mathsf{L}$)**

$$\text{Simulated } \mathcal{S}_1 \text{ sets } \mathsf{cnt}^{\gamma}_{(1,2)} \coloneqq \mathsf{cnt}^{\gamma}_{(1,0)} \coloneqq \mathsf{cnt}^{\gamma}_{(2,0)} \coloneqq 0, \quad \text{Simulated } \mathcal{S}_2 \text{ sets } \mathsf{cnt}^{\gamma}_{(2,0)} \coloneqq \mathsf{cnt}^{\gamma}_{(2,1)} \coloneqq 0.$$

5. **Pruning.** For every $(k+1)$-bit string γ, Sim simulates the pruning step as follows:

   If $\gamma \in \mathsf{HH}^{k+1}$ **then** Sim invokes the simulator of $\mathcal{F}_{\mathsf{CMP}}$ with output 1 s.t. $\mathcal{F}_{\mathsf{CMP}}$ returns 1 as output to the servers, s.t. γ is included in the list of heavy-hitting strings.

   If $\gamma \notin \mathsf{HH}^{k+1}$ **then** Sim invokes the simulator of $\mathcal{F}_{\mathsf{CMP}}$ with output 0 s.t. $\mathcal{F}_{\mathsf{CMP}}$ returns 0 as output to the servers, s.t. γ gets pruned.

   If the simulator of $\mathcal{F}_{\mathsf{CMP}}$ aborts, then Sim instructs $\mathcal{F}_{\mathsf{HH}}$ to abort at level $(\perp, k+1)$ and Sim aborts this simulated execution. Sim has successfully simulated the HH^{k+1} set. Sim repeats *"Server Computation"* steps (starting from Step 2b) on $k+1$ bit prefixes.

Output Phase.

Sim outputs $\mathsf{HH}^{\leq n}$ as the set of $\mathcal{T}$-heavy hitter strings on behalf of simulated $\mathcal{S}_1$ and $\mathcal{S}_2$, and instructs $\mathcal{F}_{\mathsf{HH}}$ to send output to the honest servers $\mathcal{S}_0$ and $\mathcal{S}_1$.

Figure 7.14: Continuing the simulation from Fig. 7.13. Simulation Algorithm against malicious corruption of server $\mathcal{S}_0$ and ℓ' clients.

HYB_2 : Same as HYB_1, except Sim extracts the corrupt client's inputs following the extraction algorithm using the pair of DPF keys. Then Sim runs Step 3d in simulated Batch-Verification, i.e., Sim aborts if 1) the client's input α_i is k-bits heavy-hitting, 2) $\alpha_i \parallel 0$ or $\alpha_1 \parallel 1$ is invalid, and 3) client evaded the Batch-Verification check for the sessions run between honest servers. The two hybrids are indistinguishable due to the verifiability property of VIDPF in the random oracle model. This occurs when a malicious client successfully evades the input extraction process of VIDPF by providing malformed VIDPF keys and yet passes the batch verification checks performed on the VIDPF proofs.

HYB_3 : Same as HYB_2, except Sim invokes $\mathcal{F}_{\mathsf{HH}}$ with the extracted inputs to obtain $\mathsf{HH}^{\leq n}$ set and simulates the $\mathcal{F}_{\mathsf{CMP}}$ functionality based on whether a prefix γ is in $\mathsf{HH}^{\leq n}$ or not. The two hybrids are indistinguishable against a corrupt server $\mathcal{S}_0$ in the $\mathcal{F}_{\mathsf{CMP}}$-model.

HYB_4 : Same as HYB_3, except Sim simulates the DPF key generation for the honest clients with input $(\alpha, (\beta_1, \ldots, \beta_n)) = (1, (1, \ldots, 1))$ and sets the counters to 0s in the aggregation step. Indistinguishable due to VIDPF input privacy. The 0-valued counters in the aggregation step are identically distributed to the actual aggregation counters since HYB_3 and HYB_4 are in the $\mathcal{F}_{\mathsf{CMP}}$-model. This is the ideal world execution of the protocol, completing our simulation algorithm.

□

7.5 Batched Consistency Check

We now present our batched consistency check π_{check} that enables two parties, P_0 and P_1, to verify the equality of lists $\mathbf{u}$ and $\mathbf{v}$ containing ℓ strings using Merkle trees. If the two lists are equal then π_{check} returns $\mathsf{ver} = 1$, else it returns $\mathsf{ver} = 0$ and a list L containing the indices of elements where the lists differ. Correctness follows from the collision resistance property of the hash function H.

π_{check}

Inputs: Party P_0 has ℓ input strings $\mathbf{u} = \{u_i\}_{i\in[\ell]}$. Party P_1 has ℓ input strings $\mathbf{v} = \{v_i\}_{i\in[\ell]}$.

Outputs: π_{check} outputs $(\mathsf{ver}, \mathsf{L})$ as follows:

- If $\mathbf{u} = \mathbf{v}$: $\mathsf{ver} \coloneqq 1$ and $\mathsf{L} \coloneqq \emptyset$,
- If $\mathbf{u} \neq \mathbf{v}$: $\mathsf{ver} \coloneqq 0$ and $\mathsf{L} \coloneqq \{i\}_{u_i \neq v_i \text{ for } i\in[\ell]}$.

$\mathsf{ver} = 1$ (resp. $\mathsf{ver} = 0$) denotes that the Merkle roots of $\mathbf{u}$ and $\mathbf{v}$ are equal (resp. unequal). List L is a list of indices where $\mathbf{u}$ and $\mathbf{v}$ differ.

Parameters: $\mathrm{H} : \{0,1\}^\kappa \to \{0,1\}^\kappa$ is a collision-resistant hash. $\mathsf{K} = \lceil \log_2 \ell \rceil$ denotes number of levels in the Merkle tree for ℓ leaves.

Algorithm:

Root Computation: Party P_0 (resp. P_1) locally computes the Merkle R_0 (resp. R_1) on $\mathbf{u}$ (resp. $\mathbf{v}$). For $b \in \{0,1\}$, party P_b performs:

- If $b = 0$: set $\mathsf{N}_0^{\mathsf{K}} \coloneqq \{\mathsf{N}_{0,i}^{\mathsf{K}}\}_{i\in[\ell]} \coloneqq \{\mathrm{H}(\mathsf{K}, i, u_i)\}_{i\in\ell}$ as the list of leaf nodes in the Merkle tree containing $\mathbf{u}$.
- If $b = 1$: set $\mathsf{N}_1^{\mathsf{K}} \coloneqq \{\mathsf{N}_{1,i}^{\mathsf{K}}\}_{i\in[\ell]} \coloneqq \{\mathrm{H}(\mathsf{K}, i, v_i)\}_{i\in\ell}$ as the list of leaf nodes in the Merkle tree containing $\mathbf{v}$.
- Initialize $\ell' \coloneqq \ell$ as the number of nodes in level K.
- For level $k \in \{\mathsf{K}-1, \mathsf{K}-2, \ldots, 1\}$:
  - Set $\ell' \coloneqq \lceil \frac{\ell'}{2} \rceil$ as the number of nodes in level k.
  - For $i \in [\ell']$: Compute list of nodes at level k by hashing the nodes at level $k+1$ as $\mathsf{N}_b^k \coloneqq \mathsf{N}_b^k \cup \mathrm{H}(k, \mathsf{N}_{b,2i}^{k+1}, \mathsf{N}_{b,2i+1}^{k+1})$.
- Set Merkle $\mathsf{R}_b \coloneqq \mathsf{N}_b^1$.

Root Verification: Parties P_0 and P_1 exchange R_0 and R_1. If $\mathsf{R}_0 = \mathsf{R}_1$ then set $\mathsf{ver} \coloneqq 1$, $\mathsf{L} \coloneqq \emptyset$, and parties output $(\mathsf{ver}, \mathsf{L})$. Else, set $\mathsf{ver} \coloneqq 0$ and continue the computation.

Unequal Leaf Identification: For $b \in \{0,1\}$, party P_b sets $\overline{\mathsf{N}}_b^1 \coloneqq \mathsf{R}_b$ as the unequal node at level 1. For level $k \in \{2, \ldots, \mathsf{K}\}$: For each unequal node $n \in \overline{\mathsf{N}}_b^{k-1}$ at level $k-1$, parties identify unequal nodes at level k:

- Party P_b fetches left and right child of n as $\mathsf{child}_b^{\mathsf{L}}$ and $\mathsf{child}_b^{\mathsf{R}}$ respectively, for $b \in \{0,1\}$.
- Parties exchange $\mathsf{child}_0^{\mathsf{L}}$, $\mathsf{child}_1^{\mathsf{L}}$, $\mathsf{child}_0^{\mathsf{R}}$ and $\mathsf{child}_1^{\mathsf{R}}$, and performs the following for $b \in \{0,1\}$:

$$\overline{\mathsf{N}}_b^k \coloneqq \overline{\mathsf{N}}_b^k \cup \mathsf{child}_b^{\mathsf{L}} \text{ if } \mathsf{child}_0^{\mathsf{L}} \neq \mathsf{child}_1^{\mathsf{L}}, \quad \overline{\mathsf{N}}_b^k \coloneqq \overline{\mathsf{N}}_b^k \cup \mathsf{child}_b^{\mathsf{R}} \text{ if } \mathsf{child}_0^{\mathsf{R}} \neq \mathsf{child}_1^{\mathsf{R}}$$

P_b possesses $\overline{\mathsf{N}}_b^{\mathsf{K}}$ as list of unequal leaf nodes. P_b sets L as the list of indices of $\overline{\mathsf{N}}_b^{\mathsf{K}}$ w.r.t. initial leaf nodes $\mathsf{N}_b^{\mathsf{K}}$ as $\mathsf{L} \coloneqq \mathsf{L} \cup \{i : \overline{\mathsf{N}}_{b,i}^{\mathsf{K}} = \mathsf{N}_{b,i}^{\mathsf{K}}\}$. Party P_b outputs $(\mathsf{ver}, \mathsf{L})$.

Figure 7.15: Equality verification of ℓ strings between two parties and identification of unequal strings.

As summarized in Fig. 7.15, π_{check} requires $\mathsf{K}+1$ rounds of communication, where $\mathsf{K} = \lceil \log_2 \ell \rceil$. The total communicated hashes are roughly $4\ell'(\log_2 \frac{\ell}{\ell'} + 2)$, where $\mathbf{u}$ and $\mathbf{v}$ differ on ℓ' elements. It can be further optimized to $2\ell'(\log_2 \frac{\ell}{\ell'} + 2)$, where only one of the parties sends its hashes instead of both. We provide a detailed analysis of the protocol in Section 7.7. In case $\ell' = 0$, then our communication is a pair of hashes.

7.6 Experimental Evaluations

We implement our heavy-hitters protocol π_{HH} in Rust and use the `tarpc` framework by Google for asynchronous Remote Procedure Calls (RPC). PLASMA is fully parallelized: all sessions in each server run in parallel and we employ parallel iterators to process multiple client requests concurrently. (We apply the same parallelization for benchmarking Poplar.) We instantiate the PRG for VIDPF using the AES-NI hardware instructions for AES encryption with a seed of $\kappa = 128$ bits. We used rings in PLASMA (instead of fields) since our checks rely on the security of VIDPF (i.e., XOR-collision resistant property that is provided by the random oracle). Conversely, the security of Poplar relies on a statistical check for the client's input validation. This check relies on the underlying group size and needs 62 bits for the statistical failure probability to be 2^{-60} for intermediate levels; for the leaves, we use the default size of a finite field of $2\kappa = 256$ bits as mentioned in Poplar.

Experiment Details. Our experiments vary the number of clients between $\ell = 10^3$ and $\ell = 10^6$ with two different bit-string sizes, $n = 64$ and $n = 256$ bits. We configured the threshold $\mathcal{T}$ to be 1% of the clients' strings, and we report the client and server costs, while empirically comparing with Poplar. Then, we compute the total monetary costs (due to runtime and communication) incurred by PLASMA servers, and we compare it with [13] (since the code of [13] is not open-source) based on the monetary cost.

Experimental Setup. We performed both LAN and WAN[1] experiments on AWS EC2 machines (c5.9xlarge) each with 36 vCPUs at 3.60 GHz. PLASMA is compiled

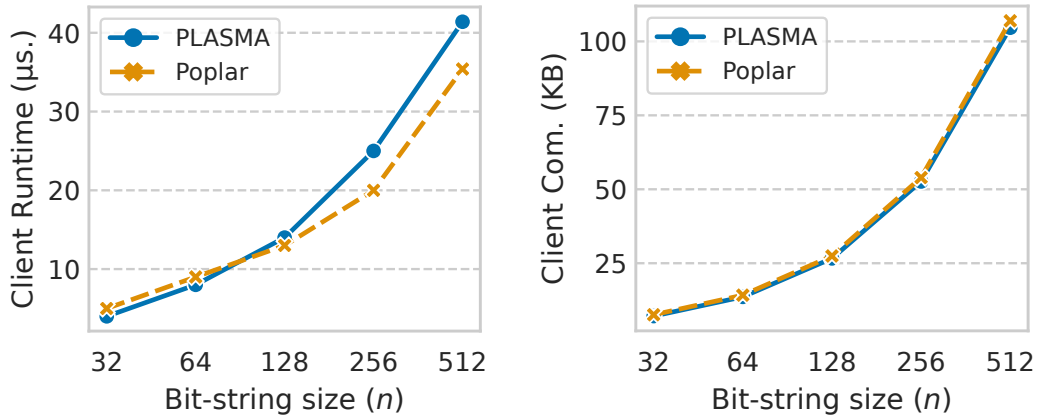
[1] We used one server in Oregon, one in Ohio, and one in N. Virginia. For Poplar, we used one in Oregon and the other one in N. Virginia.

using Rust 1.74, and client-side experiments are carried out using a standard laptop with an Intel i7-8650U CPU (1.90 GHz).

Performance Evaluation. In our experiments, our goal is to answer the following questions:

- How efficient is PLASMA for each client and server?
- How does PLASMA compare with similar works (such as Poplar) that leverage DPFs?
- How does PLASMA compare with the related works that provide similar security guarantees, such as [13]?

Client costs. The PLASMA client generates three pairs of DPF keys. Meanwhile, the Poplar client generates two pairs of DPF keys but also computes a malicious sketching operation. As a result, both PLASMA and Poplar clients are extremely fast, running in the order of $20-24$ microseconds on 256-bit inputs. A detailed comparison of client runtime can be found in Fig. 7.16 (a).



(a) Client Runtime

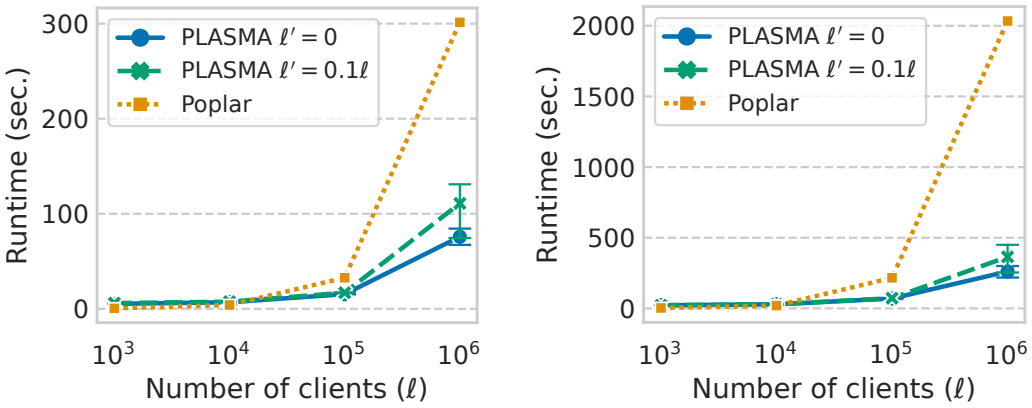
(b) Client Communication

Figure 7.16: Comparisons of client costs for PLASMA and Poplar (KB is Kilobytes and μs is microseconds).

In terms of client communication, PLASMA transmits eight DPF keys, whereas Poplar transmits four DPF keys plus the correlated randomness for the sketching operation. We observed that the clients in both protocols incur the same communication overhead, roughly around 55 KB for 256 bits. Detailed comparisons can be found in Fig. 7.16 (b).

Server costs. In this experiment, we run PLASMA with randomly distributed malicious clients and compare it with Poplar. We set the malicious clients ℓ' to be 0, 0.01, 0.1, and 0.3 of the total clients ℓ. We observed that running with $\ell' = 0.01\ell$ has slightly faster performance than 0.1ℓ, while 0.3ℓ exhibits slightly worse performance than 0.1ℓ. Still, these differences are marginal compared to the total runtime, so we opt for reporting the 0 and 0.1ℓ for the sake of making the figures clearer.

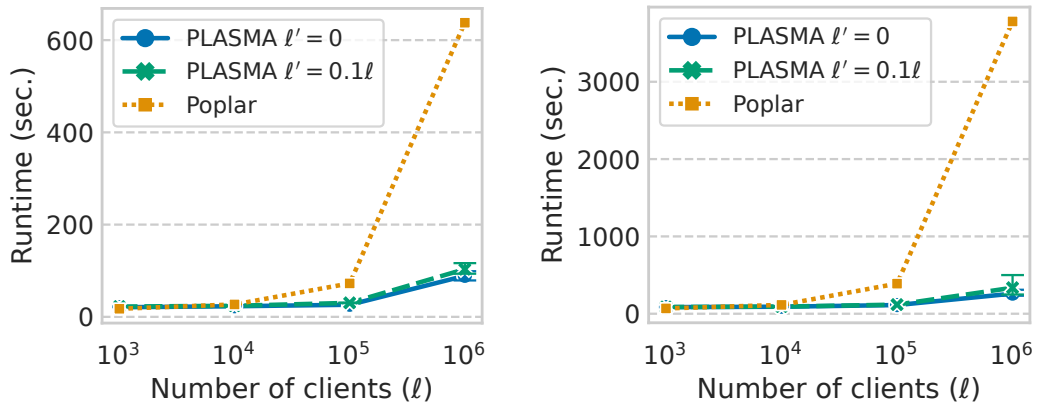
LAN Server Runtime. PLASMA outperforms Poplar in terms of server runtime by 2.7× (64 bits) and 5× (256 bits) for $\ell = 10^6$ clients and $\mathcal{T} = 1\%$ of the clients. This improvement is largely attributed to our efficient VIDPF-based client input validation. Although the presence of malicious clients has an impact on PLASMA's performance, it still remains significantly faster than Poplar as presented in Fig. 7.17. Meanwhile, Poplar servers validate clients' inputs using an expensive malicious secure sketching protocol.



(a) Bit-string size ($n = 64$) (b) Bit-string size ($n = 256$)

Figure 7.17: Server runtime (over LAN) for an increasing number of clients.

WAN Server Runtime. We benchmarked PLASMA and Poplar over WAN for $n = 64$ bits and we report our findings in Fig. 7.18. While the total latency is increased for both frameworks, we observe that the server WAN runtime for PLASMA increased by roughly 5-10% compared to server LAN runtime, whereas for Poplar the runtime increases by roughly 50%. We observe almost $5 - 10\times$ improvement in terms of server WAN runtime for PLASMA compared to Poplar since PLASMA incurs significantly less communication for $\mathcal{T} = 1\%$.



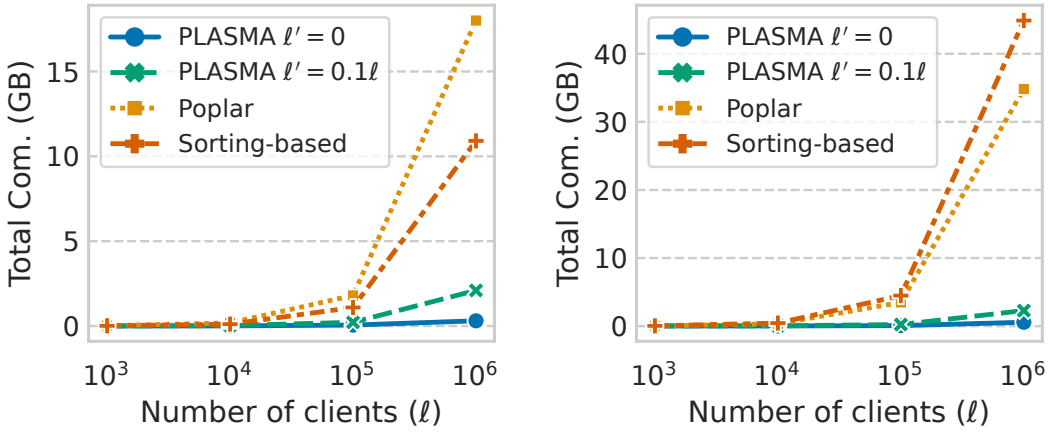
(a) Bit-string size ($n = 64$)

(b) Bit-string size ($n = 256$)

Figure 7.18: Server runtime over WAN.

Server-to-Server Communication. We compare the total communication costs incurred by all servers for an increasing number of clients, $\mathcal{T} = 1\%$, and $n = 256$ in Fig. 7.19. Poplar servers incur 35 GB of communication, whereas, PLASMA servers communicate less than 1 GB of data when considering $\ell' = 0$ and 0.1ℓ corrupt clients, hence yielding a $35\times$ improvement over Poplar. The protocol of [13] communicates 45 GB of data to compute heavy-hitters over 10^6 client submitted 256-bit inputs. This yields a $45\times$ improvement of PLASMA over [13].

Server Monetary Cost. To obtain fair comparisons between Poplar, [13], and PLASMA, we perform cumulative monetary cost analysis for a varying number of clients, assuming \$0.05/GB and \$1.53/hour. To estimate monetary costs, we run PLASMA and Poplar in a similar setup as [13] and compare it with the runtime



(a) Bit-string size $n = 64$

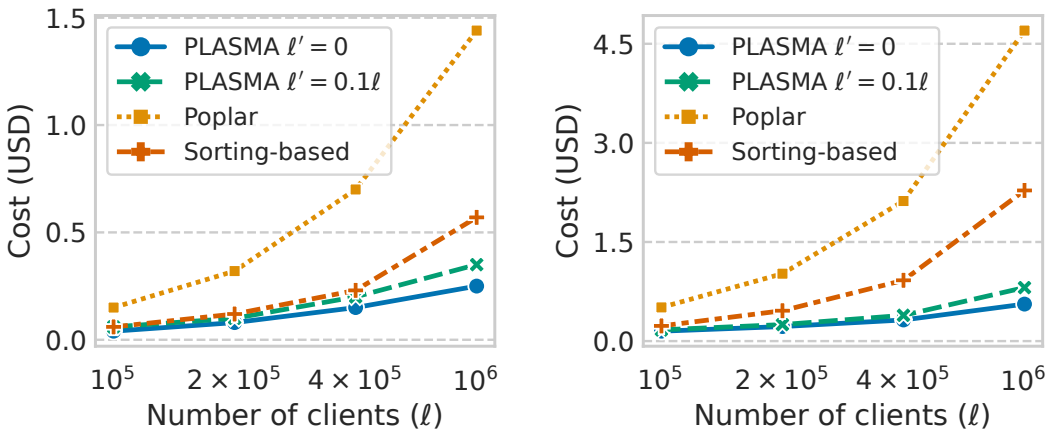
(b) Bit-string size $n = 256$

Figure 7.19: Comparisons with Poplar [49] and the sorting-based approach of [13] in terms of total server-to-server communication (in GB).

provided in [13]. Note that Poplar runs two servers while PLASMA runs three. The monetary cost incurred by Poplar is two times the cost incurred by a single Poplar server, while for PLASMA it's three times a single PLASMA server. We present our findings in Fig. 7.20 for $\mathcal{T} = 1\%$ of the clients. Computing the $\mathcal{T}$ most popular strings among 1 million clients with $n = 256$ bit strings, costs \$4.7 with Poplar, while PLASMA incurs \$0.6-\$0.9 costs for 0 to 0.1ℓ malicious clients. Meanwhile, [13] costs at least \$2.2 to perform the same task, so PLASMA yields a $2.5 - 3.5\times$ improvement over [13] despite having a $15\times$ runtime slowdown. This is largely due to the communication incurred by [13] for performing secure sorting under MPC. When considering input strings of smaller size, like $n = 64$, PLASMA is $4\times$ cheaper than Poplar and $2\times$ cheaper than [13].

Applications. We discuss two realistic applications:

Popular URLs. Each URL is represented as a 256-bit string and 10000 most popular URLs are computed among 1 million client-submitted URLs, assuming $\mathcal{T} = 1\%$. Server runtimes of PLASMA and Poplar are reported in Figs. 7.17 (b) and 7.18 (b), while the client communication costs in Figs. 7.16 (a) and (b) for $n = 256$. This benchmark is completed in under 5 minutes with less than 1 GB of data of communication



(a) Bit-string size $n = 64$

(b) Bit-string size $n = 256$

Figure 7.20: Comparisons with Poplar and the sorting-based approach of [13] in terms of total monetary cost (in USD).

for PLASMA, while Poplar servers incur more than 5× additional runtime costs and communicate 35 GB.

Popular GPS coordinates. We employ *plus codes* [167] to efficiently encode the client GPS coordinates using 64 bits. This approach uses a grid system aligned on top of the world map, assigning specific codes to each area. Areas with similar codes are located in proximity to each other and a code that is a prefix of another encompasses the area of the latter. For instance, code `87` represents the North East US region, while code `87G8` represents a part of New York City. PLASMA uses plus codes to compute the most popular locations (submitted by more than $\mathcal{T} = 1\%$ of the clients) among a set of client-provided inputs using 64-bit strings in roughly 2 minutes for 10^6 clients, as shown in Fig. 7.18 (a). Client cost is shown in Figs. 7.16 (a) and (b) for $n = 64$.

7.7 Analysis of Batched Consistency check

We recall the batched consistency check in Fig. 7.15. P_0 and P_1 hash their individual leaves and verify the equality of their Merkle tree roots R_0 and R_1. If the roots are equal then all the leaves are equal. Otherwise, the parties verify the equality of the left children and the right children of the root node. If the left (resp. right)

children are equal across the parties then the left (resp. right) subtrees are equal. If the left (resp. right) children are different, then the parties apply the above algorithm to the left (resp. right) subtree. Proceeding this way in an iterative manner down the tree, the parties identify the malformed leaves as $\overline{\mathsf{N}}_0^{\mathsf{K}}$ and $\overline{\mathsf{N}}_1^{\mathsf{K}}$ where the two trees differ. Then they match them with their initial lists of input sets $\mathbf{u}$ and $\mathbf{v}$ to identify the indices where they differ and then store those indices in L.

π_{check} requires $\mathsf{K}+1$ rounds of communication, where $\mathsf{K} = \lceil \log_2 \ell \rceil$. Next, we demonstrate that if ℓ' out of ℓ leaves differ, then the total communication is $\mathcal{O}(\ell'(\log_2 \frac{\ell}{\ell'}))$ hashes. The *Root Computation* is local and *Root Verification* communicates two hashes. During *Leaf Identification*, the parties communicate 4 hashes for each unequal node. At the root layer, only the roots are different. At the next layer, both children can differ. More generally, at layer $k \in [\mathsf{K}]$, there can be at most $\min(2^k, \ell')$ unequal nodes. The total communicated hashes are as follows:

$$
\begin{aligned}
&2 + 4 \times (\min(2^0, \ell') + \ldots + \min(2^{\lceil \log_2 \ell \rceil}, \ell')) \\
&= 2 + 4 \times (1 + 2 + \ldots 2^{\lceil \log_2 \ell' \rceil} + \ell' + \ell' + \ldots + \ell') \\
&\leq 2 + 4 \times (2\ell' + \ell' \times (\lceil \log_2 \ell \rceil - \lceil \log_2 \ell' \rceil)) \\
&\approx 8\ell' + 4\ell'(\log_2 \ell - \log_2 \ell') = 4\ell'(\log_2 \tfrac{\ell}{\ell'} + 2).
\end{aligned}
$$

We observe that the current version of π_{check} communicates roughly $4\ell'(\log_2 \frac{\ell}{\ell'} + 2)$ hashes. This can be further optimized to $2\ell'(\log_2 \frac{\ell}{\ell'} + 2)$ where only one server communicates at each level.

7.8 Heavy Hitters with different Thresholds

Our protocol allows us to consider different heavy hitter thresholds $\mathcal{T}_i$ based on some pre-agreed strings $x_i \in \mathbf{X}$ by the servers. This can be beneficial for traffic avoidance since different roads may have different traffic densities. For example, highways are busier than smaller suburban roads. The servers can take that into consideration during evaluation, and use higher $\mathcal{T}$s for highways (since there are more vehicles), and lower thresholds for smaller roads.

Different Threshold Heavy Hitters from $\mathcal{T}$-prefix count queries

Parameters: Threshold $\mathcal{T}_i \in \mathbb{N}$, for string $x_i \in \mathbf{X}$ where $|\mathbf{X}| = m$, and string length $n \in \mathbb{N}$.

Inputs: The algorithm has no explicit input. It has access to t-prefix count query oracle $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p,t)$ for securely computing t-prefix-count queries over prefix p for strings $\alpha_1,\ldots,\alpha_\ell$.

Outputs: The set of heavy-hitter strings in $\alpha_1,\alpha_2,\ldots,\alpha_\ell$.

Algorithm:

- Initialize $\mathsf{HH}^{\leq n} = \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\} \coloneqq \{\epsilon, \emptyset, \ldots, \emptyset\}$, where HH^0 contains empty string ϵ and $\mathsf{HH}^1, \ldots \mathsf{HH}^n$ are empty sets.
- Set $\mathcal{T} \coloneqq \min(\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_m)$.
- For each prefix $p \in \mathsf{HH}^k$ of length k-bits in set HH^k (where $k = 0, 1, 2, \ldots n-2$) :
  - If $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 0, \mathcal{T}) = 1$, $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \{p \parallel 0\}$.
  - If $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 1, \mathcal{T}) = 1$, $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \{p \parallel 1\}$.
- For each prefix $p \in \mathsf{HH}^{n-1}$, perform the following:
  - If $\exists x_i \in \mathbf{X}$ such that $(p \parallel 0) = x_i$ and $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 0, \mathcal{T}_i) = 1$, then set $\mathsf{HH}^n \coloneqq \mathsf{HH}^n \cup (p \parallel 0)$.
  - If $\exists x_i \in \mathbf{X}$ such that $(p \parallel 1) = x_i$ and $\Omega_{\alpha_1,\ldots,\alpha_\ell}(p \parallel 1, \mathcal{T}_i) = 1$, then set $\mathsf{HH}^n \coloneqq \mathsf{HH}^n \cup (p \parallel 1)$.
- Output $\mathcal{T}$-heavy hitters $\mathsf{HH}^{\leq n} \coloneqq \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\}$.

Figure 7.21: Algorithm for computing heavy hitters with different thresholds from $\mathcal{T}$-prefix count queries.

We present our algorithm to compute heavy-hitters with different thresholds $\mathcal{T}_i$ for string $x_i \in \mathbf{X}$ from $\mathcal{T}$-prefix oracle query in Fig. 7.21. The prefix oracle query with different thresholds can be computed using a simple modification to protocol π_{HH}, where the pruning at the leaf layer is performed based on the threshold $\mathcal{T}_i$ for a given string $x_i \in \mathbf{X}$ instead of a fixed threshold $\mathcal{T}$.

7.9 Compatibility with Differential Privacy

It is straightforward to complement PLASMA with ϵ-differential privacy techniques and ensure that the presence or absence of a single client does not reveal anything about their data [100]. In this case, running two instances of PLASMA, one with $\ell - 1$ clients and another just by adding client $\mathcal{C}$, should protect the private data of the new client from anyone observing the outputs of the two protocols. Additionally,

honest clients should not be able to be identified when a malicious server attempts to ignore honest client data to infer their inputs based on the protocol output. Therefore, PLASMA is directly compatible with the well-studied techniques from [99, 102] and can adopt a similar approach as Poplar to bound the amount of information that an adversary $\mathcal{A}$ can deduce from PLASMA's output. Like Poplar, we need to ensure that the outputs of these prefix-count oracle queries are differentially private, which can be achieved by introducing noise on the oracle's output with parameter $1/\epsilon$ from a Laplace distribution.

7.10 Concluding Remarks

In this work, we present PLASMA: a framework to privately identify the most popular strings – or heavy hitters – among a set of client inputs without revealing the client data points. Previous works for private heavy hitters, such as Poplar, consider security against malicious clients and were prone to additive attacks by a malicious server, compromising the correctness of the protocol. To address this challenge, PLASMA introduces a novel hash-based primitive, called verifiable incremental distributed point functions, which allows the servers to validate client inputs using inexpensive operations. Additionally, we introduce a new batched consistency check that uses Merkle trees to validate multiple client sessions in a batch. This drastically reduces the concrete server-to-server communication, incurred during the heavy-hitters computation.

Chapter 8

CONCLUSION

In conclusion, this thesis has delved into the realm of Privacy Enhancing Technologies (PETs) and various real-world applications of PETs in safeguarding sensitive data. More specifically, we focus on private and verifiable computation; i.e., to perform meaningful computations while both protecting data privacy and also providing guarantees that the computation was executed correctly.

We started by introducing Zilch, our framework for deploying transparent Zero-Knowledge Proofs (ZKP) from high-level Java-like code. Zilch incorporates a novel cross-compiler from an object-oriented Java-like language tailored to ZKPs as well as a powerful API that enables integration of ZKPs within existing C/C++ programs. Zilch is transparent, in that it does not need a trusted setup, and fosters usability by making ZKPs more accessible and efficient.

Next, we utilized Zilch to address a critical concern in the Integrated Circuits (IC) industry, i.e., to verify Intellectual Properties (IP) without compromising the privacy of circuit implementations. The first work in this area, dubbed Pythia, allows third-party intellectual property (3PIP) vendors to prove to system integrators the functional properties of their circuit designs while protecting the privacy of the circuit implementations. We continued this line of work with zk-Sherlock, a framework that enables 3PIP vendors to prove an intellectual property (IP) design is free of hardware Trojans (i.e., malicious circuit modifications that alter device functionalities or leak sensitive information when triggered) without disclosing the corresponding netlist. Together, Pythia and zk-Sherlock can mitigate the threat of IP piracy in the IC industry.

In our next line of work, we focused on computing private statistics while ensuring data privacy and integrity against malicious inputs. We introduced two protocols

for privacy-preserving statistics, Masquerade and PLASMA. Both works focus on data aggregation and statistics based on inputs from multiple participants while preserving the privacy of the participant data. Additionally, as participants may misbehave and either try to glean information about other participants' data or attempt to affect the correctness of the computation, Masquerade and PLASMA provide verifiable guarantees that such malicious behavior is detected.

This dissertation has contributed both theoretical and practical solutions to real-world challenges, from protecting sensitive IP designs in the IC industry to preserving integrity and privacy in large-scale statistics. It is evident that PETs hold immense promise in ensuring the security and privacy of data in an increasingly digital world. The path forward involves further refinement, collaboration, and the integration of these techniques into a wide range of applications.

BIBLIOGRAPHY

[1] Accountability Act. Health insurance portability and accountability act of 1996. *Public law*, 104:191, 1996.

[2] Carlisle Adams and Steve Lloyd. *Understanding PKI: concepts, standards, and deployment considerations*. Addison-Wesley Professional, 2003.

[3] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[4] Yousra Alkabani and Farinaz Koushanfar. Consistency-based characterization for ic trojan detection. In *2009 ICCAD*, pages 123–127, 2009.

[5] Abdelrahaman Aly, K Cong, D Cozzo, M Keller, E Orsini, D Rotaru, O Scherer, P Scholl, N Smart, T Tanguy, et al. Scale–mamba v1. 12: Documentation, 2021.

[6] Abdelrahaman Aly, Marcel Keller, Dragos Rotaru, Peter Scholl, Nigel Smart, and Tim Wood. SCALE–MAMBA Documentation, 2021.

[7] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[8] Erik Anderson, Melissa Chase, F. Betul Durak, Esha Ghosh, Kim Laine, and Chenkai Weng. Aggregate measurement via oblivious shuffling. Cryptology ePrint Archive, Report 2021/1490, 2021. https://eprint.iacr.org/2021/1490.

[9] Apple and Google. Exposure Notification Privacy-preserving Analytics (ENPA) white paper, 2021.

[10] Diego F. Aranha, Paulo S. L. M. Barreto, Geovandro C. C. F. Pereira, and Jefferson E. Ricardini. A note on high-security general-purpose elliptic curves. Cryptology ePrint Archive, Report 2013/647, 2013. https://eprint.iacr.org/2013/647.

[11] Sanjeev Arora et al. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998.

[12] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of NP. *Journal of the ACM (JACM)*, 45(1):70–122, 1998.

[13] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 125–138. ACM Press, November 2022.

[14] N. Asokan. Hardware-assisted trusted execution environments: Look back, look ahead. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, page 1687. ACM Press, November 2019.

[15] László Babai. Trading group theory for randomness. In *17th ACM STOC*, pages 421–429. ACM Press, May 1985.

[16] Michael Backes, Manuel Barbosa, Dario Fiore, and Raphael M. Reischuk. ADSNARK: Nearly practical and privacy-preserving proofs on authenticated data. In *2015 IEEE Symposium on Security and Privacy*, pages 271–286. IEEE Computer Society Press, May 2015.

[17] Michael Backes, Dario Fiore, and Raphael M. Reischuk. Verifiable delegation of computation on outsourced data. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 863–874. ACM Press, November 2013.

[18] Elaine B. Barker, William C. Barker, William E. Burr, W. Timothy Polk, and Miles E. Smid. *Recommendation for Key Management: Part 1 Rev. 5*. National Institute of Standards and Technology, Technology Administration, Gaithersburg, MD, USA, 2007.

[19] Raef Bassily, Kobbi Nissim, Uri Stemmer, and Abhradeep Guha Thakurta. Practical locally private heavy hitters. *Advances in Neural Information Processing Systems*, 30:1–32, 2017.

[20] Olivier Baudron, Pierre-Alain Fouque, David Pointcheval, Jacques Stern, and Guillaume Poupard. Practical multi-candidate election system. In Ajay D. Kshemkalyani and Nir Shavit, editors, *20th ACM PODC*, pages 274–283. ACM, August 2001.

[21] Ray Beaulieu et al. The SIMON and SPECK lightweight block ciphers. In *DAC*, pages 1–6. ACM/EDAC/IEEE, 2015.

[22] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.

[23] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[24] Peter A Beerel and Teresa H-Y Meng. Automatic gate-level synthesis of speed-independent circuits. In *ICCAD*, pages 581–586. IEEE/ACM, 1992.

[25] Amos Beimel. Secret-sharing schemes: A survey. In *International Conference on Coding and Cryptology*, pages 11–46, Berlin, Heidelberg, 2011. Springer.

[26] Amos Beimel and Benny Chor. Universally ideal secret sharing schemes (preliminary version). In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 183–195. Springer, Heidelberg, August 1993.

[27] James Bell, Adrià Gascón, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, Mariana Raykova, and Phillipp Schoppmann. Distributed, private, sparse histograms in the two-server model. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 307–321. ACM Press, November 2022.

[28] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993.

[29] Eli Ben-Sasson, Iddo Bentov, Alessandro Chiesa, Ariel Gabizon, Daniel Genkin, Matan Hamilis, Evgenya Pergament, Michael Riabzev, Mark Silberstein, Eran Tromer, and Madars Virza. Computational integrity with a public random string from quasi-linear PCPs. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 551–579. Springer, Heidelberg, April / May 2017.

[30] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018*, volume 107 of *LIPIcs*, pages 14:1–14:17. Schloss Dagstuhl, July 2018.

[31] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 701–732. Springer, Heidelberg, August 2019.

[32] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.

[33] 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, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Heidelberg, August 2013.

[34] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Heidelberg, May 2019.

[35] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 31–60. Springer, Heidelberg, October / November 2016.

[36] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 276–294. Springer, Heidelberg, August 2014.

[37] 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, *USENIX Security 2014*, pages 781–796. USENIX Association, August 2014.

[38] Eli Ben-Sasson et al. libSTARK: a C++ library for zk-STARK systems. https://github.com/elibensasson/libSTARK, 2018. [Online].

[39] Daniel J Bernstein. Introduction to post-quantum cryptography. In *Post-Quantum Cryptography*, pages 1–14. Springer, 2009.

[40] Barry Bishop and Florian Fischer. Iris-integrated rule inference system. In *International Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008)*. sn, 2008.

[41] Nir Bitansky et al. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, pages 326–349. ACM, 2012.

[42] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong Privacy for Analytics in the Crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 441–459, New York, NY, USA, 2017. Association for Computing Machinery.

[43] Manuel Blum. Coin flipping by telephone. In *Proc. IEEE Spring COMPCOM*, pages 133–137, 1982.

[44] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE2: A high-throughput framework for neural network inference on encrypted data. Cryptology ePrint Archive, Report 2019/947, 2019. https://eprint.iacr.org/2019/947.

[45] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206. Springer, Heidelberg, October 2008.

[46] Jonas Böhler and Florian Kerschbaum. Secure multi-party computation of differentially private heavy hitters. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2361–2377. ACM Press, November 2021.

[47] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1175–1191. ACM Press, October / November 2017.

[48] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 67–97. Springer, Heidelberg, August 2019.

[49] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy*, pages 762–776. IEEE Computer Society Press, May 2021.

[50] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Arithmetic sketching. In *CRYPTO 2023, Part I*, LNCS, pages 171–202. Springer, Heidelberg, August 2023.

[51] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 327–357. Springer, Heidelberg, May 2016.

[52] Fabrice Boudot. Efficient proofs that a committed number lies in an interval. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 431–444. Springer, Heidelberg, May 2000.

[53] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019. https://eprint.iacr.org/2019/1021.

[54] Joan Boyar, Carsten Lund, and René Peralta. On the communication complexity of zero-knowledge proofs. *Journal of Cryptology*, 6(2):65–85, June 1993.

[55] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Heidelberg, April 2015.

[56] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[57] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

[58] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of computer and system sciences*, 37(2):156–189, 1988.

[59] Franc Brglez. A neural netlist of 10 combinational benchmark circuits. *IEEE ISCAS: Special Session on ATPG and Fault Simulation*, pages 151–158, 1985.

[60] Franc Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. In *ISCAS*, pages 1929–1934. IEEE, 1989.

[61] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.

[62] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 677–706. Springer, Heidelberg, May 2020.

[63] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas A. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security 2010*, pages 223–240. USENIX Association, August 2010.

[64] Andrew E Caldwell et al. Effective iterative techniques for fingerprinting design IP. *IEEE TCAD*, 23(2):208–215, 2004.

[65] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2075–2092. ACM Press, November 2019.

[66] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.

[67] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[68] Benjamin Case, Richa Jain, Alex Koshelev, Andy Leiserson, Daniel Masny, Ben Savage, Erik Taubeneck, Martin Thomson, and Taiki Yamaguchi. Interoperable Private Attribution: A Distributed Attribution and Aggregation Protocol. Cryptology ePrint Archive, Report 2023/437, 2023. https://eprint.iacr.org/2023/437.

[69] Encarnacin Castillo et al. IPP@HDL: Efficient Intellectual Property Protection Scheme for IP Cores. *IEEE TVLSI*, 15(5):578–591, 2007.

[70] Ann Cavoukian, Jules Polonetsky, and Christopher Wolf. Smart privacy for the smart grid: embedding privacy into the design of electricity conservation. *Identity in the Information Society*, 3(2):275–294, 2010.

[71] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. Privacy-preserving stream aggregation with fault tolerance. In Angelos D. Keromytis, editor, *FC 2012*, volume 7397 of *LNCS*, pages 200–214. Springer, Heidelberg, February / March 2012.

[72] Pankaj Chauhan et al. Verifying IP-core based system-on-chip designs. In *ASIC/-SOC*, pages 27–31. IEEE, 1999.

[73] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Heidelberg, December 2017.

[74] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Heidelberg, August 2018.

[75] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020.

[76] Alessandro Chiesa, Peter Manohar, and Nicholas Spooner. Succinct arguments in the quantum random oracle model. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 1–29. Springer, Heidelberg, December 2019.

[77] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 377–408. Springer, Heidelberg, December 2017.

[78] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020.

[79] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 670–699. Springer, Heidelberg, December 2021.

[80] I. Ciofi et al. Impact of Wire Geometry on Interconnect RC and Circuit Delay. *IEEE Transactions on Electron Devices*, 63(6):2488–2496, 2016.

[81] F. Corno, M.S. Reorda, and G. Squillero. RT-level ITC'99 benchmarks and first ATPG results. *IEEE Design Test of Computers*, 17(3):44–53, 2000.

[82] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 430–448. Springer, Heidelberg, August 2005.

[83] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 259–282, USA, 2017. USENIX Association.

[84] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, pages 253–270. IEEE Computer Society Press, May 2015.

[85] Scott E. Coull, Charles V. Wright, Fabian Monrose, Michael P. Collins, and Michael K. Reiter. Playing devil's advocate: Inferring sensitive information from anonymized network traces. In *NDSS 2007*. The Internet Society, February / March 2007.

[86] Jonathan Cruz et al. An automated configurable trojan insertion framework for dynamic trust benchmarks. In *DATE*, pages 1598–1603, 2018.

[87] Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 501–520. Springer, Heidelberg, August 2006.

[88] Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 119–136. Springer, Heidelberg, February 2001.

[89] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[90] George Danezis, Cédric Fournet, Markulf Kohlweiss, and Santiago Zanella-Béguelin. Smart meter aggregation via secret-sharing. In *Proceedings of the first ACM workshop on Smart energy grid security*, pages 75–80, 2013.

[91] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Chet: An optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 142–156, New York, NY, USA, 2019. Association for Computing Machinery.

[92] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. In *2022 IEEE Symposium on Security and Privacy*, pages 2450–2468. IEEE Computer Society Press, May 2022.

[93] Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann. Verifiable Distributed Aggregation Functions. *PoPETs*, 2023(4):578–592, July 2023.

[94] Leo de Castro and Antigoni Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 150–179. Springer, Heidelberg, May / June 2022.

[95] Anil Deshpande. Verification of IP-Core based SoC's. In *ISQED*, pages 433–436. IEEE, 2008.

[96] John R Douceur. The Sybil Attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[97] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 201–210. JMLR.org, 2016.

[98] Cynthia Dwork. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation*, pages 1–19. Springer, 2008.

[99] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 486–503. Springer, Heidelberg, May / June 2006.

[100] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 265–284. Springer, Heidelberg, March 2006.

[101] Jacob Eberhardt and Stefan Tai. ZoKrates - Scalable Privacy-Preserving Off-Chain Computations. In *iThings/GreenCom/CPSCom/SmartData*, pages 1084–1091. IEEE, 2018.

[102] Tariq Elahi, George Danezis, and Ian Goldberg. PrivEx: Private collection of traffic statistics for anonymous communication networks. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 1068–1079. ACM Press, November 2014.

[103] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 1054–1067. ACM Press, November 2014.

[104] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Found. Trends Priv. Secur.*, 2(2–3):70–246, dec 2018.

[105] Giulia Fanti, Vasyl Pihur, and Úlfar Erlingsson. Building a RAPPOR with the Unknown: Privacy-Preserving Learning of Associations and Data Dictionaries. *Proc. Priv. Enhancing Technol.*, 2016(3):41–61, 2016.

[106] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

[107] Matthias Fitzi, Martin Hirt, and Ueli M. Maurer. Trading correctness for privacy in unconditional multi-party computation (extended abstract). In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 121–136. Springer, Heidelberg, August 1998.

[108] Lars Folkerts, Charles Gouert, and Nektarios Georgios Tsoutsos. REDsec: Running encrypted DNNs in seconds. Cryptology ePrint Archive, Report 2021/1100, 2021. https://eprint.iacr.org/2021/1100.

[109] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, April / May 2017.

[110] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[111] Steven D. Galbraith, Chris Heneghan, and James F. McKee. Tunable balancing of RSA. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP 05*, volume 3574 of *LNCS*, pages 280–292. Springer, Heidelberg, July 2005.

[112] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 465–482. Springer, Heidelberg, August 2010.

[113] 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, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

[114] Craig Gentry and Dan Boneh. *A fully homomorphic encryption scheme*. Stanford university Stanford, 2009.

[115] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 75–92. Springer, Heidelberg, August 2013.

[116] Abhijit Ghosh et al. Estimation of average switching activity in combinational and sequential circuits. In *DAC*, volume 29, pages 253–269. ACM/EDAC/IEEE, 1992.

[117] Thanos Giannopoulos and Dimitris Mouris. Privacy preserving medical data analytics using secure multi party computation. an end-to-end use case. Master's thesis, National and Kapodistrian University of Athens, 2018.

[118] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Heidelberg, May 2014.

[119] Oded Goldreich and Ariel Kahan. How to construct constant-round zero-knowledge proof systems for NP. *Journal of Cryptology*, 9(3):167–190, June 1996.

[120] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[121] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):691–729, July 1991.

[122] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, December 1994.

[123] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 113–122. ACM Press, May 2008.

[124] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[125] Charles Gouert and Nektarios Georgios Tsoutsos. ROMEO: Conversion and Evaluation of HDL Designs in the Encrypted Domain. In *DAC*, pages 1–6. ACM/EDAC/IEEE, 2020.

[126] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 321–340. Springer, Heidelberg, December 2010.

[127] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.

[128] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-SNARKs. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 698–728. Springer, Heidelberg, August 2018.

[129] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In Jonathan Katz and Hovav

Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 581–612. Springer, Heidelberg, August 2017.

[130] Jayavardhana Gubbi et al. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[131] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved RNS variant of the BFV homomorphic encryption scheme. In Mitsuru Matsui, editor, *CT-RSA 2019*, volume 11405 of *LNCS*, pages 83–105. Springer, Heidelberg, March 2019.

[132] Justin Hsu, Sanjeev Khanna, and Aaron Roth. Distributed Private Heavy Hitters. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part I*, ICALP'12, page 461–472, Berlin, Heidelberg, 2012. Springer-Verlag.

[133] Tsung-Wei Huang and Martin DF Wong. OpenTimer: A high-performance timing analysis tool. In *ICCAD*, pages 895–902. IEEE, 2015.

[134] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. On Deploying Secure Computing: Private Intersection-Sum-with-Cardinality. In *EuroS&P*, pages 370–389, Genoa, Italy, 2020. IEEE.

[135] Jim Isaak and Mina J Hanna. User data privacy: Facebook, Cambridge Analytica, and privacy protection. *Computer*, 51(8):56–59, 2018.

[136] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[137] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 572–591. Springer, Heidelberg, August 2008.

[138] Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Vogue: Faster computation of private heavy hitters. Cryptology ePrint Archive, Paper 2022/1561, 2022. https://eprint.iacr.org/2022/1561.

[139] Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Vogue: Faster computation of private heavy hitters. Cryptology ePrint Archive, Report 2022/1561, 2022. https://eprint.iacr.org/2022/1561.

[140] Yier Jin and Yiorgos Makris. Hardware trojan detection using path delay fingerprint. In *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, pages 51–57. IEEE, 2008.

[141] Yier Jin and Yiorgos Makris. Proof carrying-based information flow tracking for data secrecy protection and hardware trust. In *VTS*, pages 252–257. IEEE, 2012.

[142] Marc Joye and Benoît Libert. A scalable scheme for privacy-preserving aggregation of time-series data. In Ahmad-Reza Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 111–125. Springer, Heidelberg, April 2013.

[143] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 114–148, 2021.

[144] Andrew B Kahng et al. Watermarking techniques for intellectual property protection. In *DAC*, pages 776–781. IEEE/ACM, 1998.

[145] Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multiparty computation. Cryptology ePrint Archive, Report 2011/272, 2011. https://eprint.iacr.org/2011/272.

[146] Ramesh Karri et al. Trustworthy hardware: Identifying and classifying hardware Trojans. *IEEE Computer*, 43(10):39–46, 2010.

[147] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2014.

[148] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020.

[149] Brian Keng and Andreas Veneris. Path-Directed Abstraction and Refinement for SAT-Based Design Debugging. *IEEE TCAD*, 32(10):1609–1622, 2013.

[150] Christoph Kern and Mark R Greenstreet. Formal verification in hardware design: a survey. *ACM TODAES*, 4(2):123–193, 1999.

[151] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *24th ACM STOC*, pages 723–732. ACM Press, May 1992.

[152] Miran Kim, Xiaoqian Jiang, Kristin Lauter, Elkhan Ismayilzada, and Shayan Shams. Secure human action recognition by encrypted neural network inference. *Nature Communications*, 13(1):4799, 2022.

[153] Tommy Koens, Coen Ramaekers, and Cees Van Wijk. Efficient Zero-Knowledge Range Proofs in Ethereum. Technical report, Technical Report, 2018.

[154] Charalambos Konstantinou, Anastasis Keliris, and Michail Maniatakos. Privacy-preserving functional IP verification utilizing fully homomorphic encryption. In *DATE*, pages 333–338. EDAA, 2015.

[155] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, Mahmoud F. Sayed, Elaine Shi, and Nikos Triandopoulos. TRUESET: Faster verifiable set computations. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 765–780. USENIX Association, August 2014.

[156] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, and Dawn Song. MIRAGE: Succinct arguments for randomized algorithms with applications to universal zk-SNARKs. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 2129–2146. USENIX Association, August 2020.

[157] Ahmed E. Kosba, Charalampos Papamanthou, and Elaine Shi. xJsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy*, pages 944–961. IEEE Computer Society Press, May 2018.

[158] Ted Krovetz. Message authentication on 64-bit architectures. In Eli Biham and Amr M. Youssef, editors, *SAC 2006*, volume 4356 of *LNCS*, pages 327–341. Springer, Heidelberg, August 2007.

[159] Fadi J Kurdahi and Alice C Parker. PLEST: a program for area estimation of VLSI integrated circuits. In *DAC*, pages 467–473. ACM/EDAC/IEEE, 1986.

[160] Iraklis Leontiadis, Kaoutar Elkhiyaoui, and Refik Molva. Private and dynamic time-series data aggregation with trust relaxation. In Dimitris Gritzalis, Aggelos Kiayias, and Ioannis G. Askoxylakis, editors, *CANS 14*, volume 8813 of *LNCS*, pages 305–320. Springer, Heidelberg, October 2014.

[161] Iraklis Leontiadis, Kaoutar Elkhiyaoui, Melek Önen, and Refik Molva. PUDA - privacy and unforgeability for data aggregation. In Michael Reiter and David Naccache, editors, *CANS 15*, LNCS, pages 3–18. Springer, Heidelberg, December 2015.

[162] Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Karn Seth, and Ni Trieu. Private join and compute from PIR with default. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part II*, volume 13091 of *LNCS*, pages 605–634. Springer, Heidelberg, December 2021.

[163] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[164] Yehuda Lindell. Fast secure two-party ECDSA signing. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 613–644. Springer, Heidelberg, August 2017.

[165] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, Heidelberg, May 2007.

[166] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society Press, May 2015.

[167] Google LLC. Open Location Code. https://github.com/google/open-location-code, 2019.

[168] Michael Luby and Charles Rackoff. How to construct pseudo-random permutations from pseudo-random functions (abstract). In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, page 447. Springer, Heidelberg, August 1986.

[169] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. In *31st FOCS*, pages 2–10. IEEE Computer Society Press, October 1990.

[170] Prince Mahajan et al. Depot: Cloud storage with minimal trust. *ACM TOCS*, 29(4):12, 2011.

[171] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. In Nikita Borisov and Claudia Díaz, editors, *FC 2021, Part I*, volume 12674 of *LNCS*, pages 249–270. Springer, Heidelberg, March 2021.

[172] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In Matt Blaze, editor, *USENIX Security 2004*, pages 287–302. USENIX Association, August 2004.

[173] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2111–2128. ACM Press, November 2019.

[174] Oleg Mazonka, Nektarios Georgios Tsoutsos, and Michail Maniatakos. Cryptoleq: A heterogeneous abstract machine for encrypted and unencrypted computation. *IEEE Transactions on Information Forensics and Security*, 11(9):2123–2138, 2016.

[175] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[176] José Monteiro et al. Estimation of average switching activity in combinational logic circuits using symbolic simulation. *IEEE TCAD*, 16(1):121–127, 1997.

[177] Gabe Moretti et al. Your Core – My Problem? Integration and Verification of IP. In *DAC*, pages 170–171. IEEE/ACM, 2001.

[178] Dimitris Mouris, Charles Gouert, and Nektarios Georgios Tsoutsos. Privacy-Preserving IP Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(7):2010–2023, 2021.

[179] Dimitris Mouris, Charles Gouert, and Nektarios Georgios Tsoutsos. zk-Sherlock: Exposing Hardware Trojans in Zero-Knowledge. In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 170–175, 2022.

[180] Dimitris Mouris, Daniel Masny, Ni Trieu, Shubho Sengupta, Prasad Buddhavarapu, and Benjamin M Case. Delegated Private Matching for Compute. *Proceedings on Privacy Enhancing Technologies*, 2024(2):1–24, July 2024.

[181] Dimitris Mouris, Pratik Sarkar, and Nektarios Georgios Tsoutsos. PLASMA: Private, lightweight aggregated statistics against malicious adversaries with full security. Cryptology ePrint Archive, Report 2023/080, 2023. https://eprint.iacr.org/2023/080.

[182] Dimitris Mouris and Nektarios Georgios Tsoutsos. Pythia: Intellectual Property Verification in Zero-Knowledge. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[183] Dimitris Mouris and Nektarios Georgios Tsoutsos. Masquerade: Verifiable multiparty aggregation with secure multiplicative commitments. Cryptology ePrint Archive, Report 2021/1370, 2021. https://eprint.iacr.org/2021/1370.

[184] Dimitris Mouris and Nektarios Georgios Tsoutsos. Zilch: A Framework for Deploying Transparent Zero-Knowledge Proofs. *IEEE Transactions on Information Forensics and Security*, 16:3269–3284, 2021.

[185] Dimitris Mouris, Nektarios Georgios Tsoutsos, and Michail Maniatakos. TERMinator Suite: Benchmarking Privacy-Preserving Architectures. *IEEE Computer Architecture Letters*, 17(2):122–125, 2018.

[186] David Naccache and Jacques Stern. A new public key cryptosystem based on higher residues. In Li Gong and Michael K. Reiter, editors, *ACM CCS 98*, pages 59–66. ACM Press, November 1998.

[187] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[188] Moni Naor, Benny Pinkas, and Eyal Ronen. How to (not) share a password: Privacy preserving protocols for finding heavy hitters with adversarial behavior. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1369–1386. ACM Press, November 2019.

[189] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *2008 IEEE Symposium on Security and Privacy*, pages 111–125. IEEE Computer Society Press, May 2008.

[190] Neha Narula, Willy Vasquez, and Madars Virza. zkLedger: Privacy-Preserving Auditing for Distributed Ledgers. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 65–80, 2018.

[191] Mahadevamurty Nemani and Farid N Najm. High-level area and power estimation for VLSI circuits. *IEEE TCAD*, 18(6):697–713, 1999.

[192] MA Nourian, Mahdi Fazeli, and David Hély. Hardware trojan detection using an advised genetic algorithm based logic testing. *JETTA*, 34(4):461–470, 2018.

[193] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[194] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg, May 1999.

[195] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013.

[196] David A Patterson and John L Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

[197] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Heidelberg, August 1992.

[198] Antigoni Polychroniadou, Gilad Asharov, Benjamin E. Diamond, Tucker Balch, Hans Buehler, Richard Hua, Suwen Gu, Greg Gimler, and Manuela Veloso. Prime Match: A Privacy-Preserving Inventory Matching System, 2023.

[199] John Proos and Christof Zalka. Shor's Discrete Logarithm Quantum Algorithm for Elliptic Curves. *Quantum Info. Comput.*, 3(4):317–344, July 2003.

[200] Zhan Qin, Yin Yang, Ting Yu, Issa Khalil, Xiaokui Xiao, and Kui Ren. Heavy hitter estimation over set-valued data with local differential privacy. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 192–203. ACM Press, October 2016.

[201] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[202] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, February 1978.

[203] Masoud Rostami et al. Hardware security: Threat models and metrics. In *ICCAD*, pages 819–823. IEEE/ACM, 2013.

[204] Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. A primer on hardware security: Models, methods, and metrics. *Proceedings of the IEEE*, 102(8):1283–1295, 2014.

[205] Jarrod A Roy, Farinaz Koushanfar, and Igor L Markov. EPIC: Ending piracy of integrated circuits. In *DATE*, pages 1069–1074. ACM, 2008.

[206] Hassan Salmani, Mohammad Tehranipoor, and Ramesh Karri. On design vulnerability analysis and trust benchmarks development. In *ICCD*, pages 471–474. IEEE, 2013.

[207] Fred B Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems (TOCS)*, 2(2):145–154, 1984.

[208] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990.

[209] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *ACNS 16*, volume 9696 of *LNCS*, pages 346–366. Springer, Heidelberg, June 2016.

[210] Srinath Setty, Andrew J Blumberg, and Michael Walfish. Toward practical and unconditional verification of remote computations. In *USENIX HotOS*, volume 13, pages 29–29, 2011.

[211] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[212] Adi Shamir. IP = PSPACE (interactive proof = polynomial space). In *FOCS*, pages 11–15. IEEE, 1990.

[213] Adi Shamir, Ronald L Rivest, and Leonard M Adleman. Mental poker. In *The mathematical gardner*, pages 37–43. Springer, 1981.

[214] Elaine Shi, T.-H. Hubert Chan, Eleanor G. Rieffel, Richard Chow, and Dawn Song. Privacy-preserving aggregation of time-series data. In *NDSS 2011*. The Internet Society, February 2011.

[215] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994.

[216] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[217] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. https://eprint.iacr.org/2004/332.

[218] Michael Sipser et al. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[219] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In *International Datalog 2.0 Workshop*, pages 245–251. Springer, 2010.

[220] Robert F Sproull and Ivan E Sutherland. Logical effort: Designing for speed on the back of an envelope. *IEEE Advanced Research in VLSI*, 9:219, 1991.

[221] Manfred Stadler et al. Functional verification of intellectual properties (IP): a simulation-based solution for an application-specific instruction-set processor. In *IEEE ITC*, pages 414–420, 1999.

[222] Succinct Computational Integrity and Privacy Research (SCIPR Lab). libsnark. https://github.com/scipr-lab/libsnark, 2014. [Online].

[223] Hassan Takabi, James BD Joshi, and Gail-Joon Ahn. Security and privacy challenges in cloud computing environments. *IEEE S&P*, 8(6):24–31, 2010.

[224] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design & Test of Computers*, 27(1):10–25, 2010.

[225] Mohammad Tehranipoor and Cliff Wang. *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.

[226] Randy Torrance and Dick James. The state-of-the-art in IC reverse engineering. In *CHES*, pages 363–381. Springer, 2009.

[227] Nektarios Georgios Tsoutsos, Charalambos Konstantinou, and Michail Maniatakos. Advanced techniques for designing stealthy hardware trojans. In *DAC*, pages 1–4. ACM, 2014.

[228] Nektarios Georgios Tsoutsos and Michail Maniatakos. Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation. *IEEE TETC*, 2(1):81–93, 2013.

[229] Nektarios Georgios Tsoutsos and Michail Maniatakos. The HEROIC framework: Encrypted computation without shared keys. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(6):875–888, 2015.

[230] Nektarios Georgios Tsoutsos and Michail Maniatakos. Efficient Detection for Malicious and Random Errors in Additive Encrypted Computation. *IEEE Transactions on Computers*, 67(1):16–31, 2017.

[231] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A Bandwidth-Efficient Distributed ORAM for 2- and 3-Party Computation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3907–3924, Anaheim, CA, August 2023. USENIX Association.

[232] Adithya Vadapalli, Kyle Storrier, and Ryan Henry. Sabre: Sender-anonymous messaging with fast audits. In *2022 IEEE Symposium on Security and Privacy*, pages 1953–1970. IEEE Computer Society Press, May 2022.

[233] William Vickrey. Optimal auctions. *The American Economic Review*, 71(3):381–392, 1981.

[234] Paul Voigt and Axel Von dem Bussche. The EU general data protection regulation (GDPR). *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 10(3152676):10–5555, 2017.

[235] Riad S Wahby et al. Reference implementation of Hyrax and Bulletproofs. https://github.com/hyraxZK/hyraxZK, 2018. [Online].

[236] Riad S. Wahby, Max Howald, Siddharth J. Garg, abhi shelat, and Michael Walfish. Verifiable ASICs. In *2016 IEEE Symposium on Security and Privacy*, pages 759–778. IEEE Computer Society Press, May 2016.

[237] Riad S. Wahby, Ye Ji, Andrew J. Blumberg, abhi shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2071–2086. ACM Press, October / November 2017.

[238] 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 *NDSS 2015*. The Internet Society, February 2015.

[239] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy*, pages 926–943. IEEE Computer Society Press, May 2018.

[240] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. FANCI: identification of stealthy malicious logic using boolean functional analysis. In *CCS*, pages 697–708. ACM, 2013.

[241] Michael Walfish and Andrew J Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2):74–84, 2015.

[242] Peter Wegner. A technique for counting ones in a binary computer. *Communications of the ACM*, 3(5):322, 1960.

[243] Neil HE Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Pearson Education India, 2015.

[244] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys - A Free Verilog Synthesis Suite. In *Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[245] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 733–764. Springer, Heidelberg, August 2019.

[246] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

[247] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[248] ZCash. Parameter Generation Ceremony and Destruction of Toxic Waste. https://z.cash/technology/paramgen/, 2016.

[249] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *2020 IEEE Symposium on Security and Privacy*, pages 859–876. IEEE Computer Society Press, May 2020.

[250] Jie Zhang et al. VeriTrust: Verification for Hardware Trust. *IEEE TCAD*, 34(7):1148–1161, 2015.

[251] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy*, pages 908–925. IEEE Computer Society Press, May 2018.

[252] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, and Dan Meng. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *2020 IEEE Symposium on Security and Privacy*, pages 1450–1465. IEEE Computer Society Press, May 2020.

[253] Wennan Zhu, Peter Kairouz, Brendan McMahan, Haicheng Sun, and Wei Li. Federated Heavy Hitters Discovery with Differential Privacy. In Silvia Chiappa and Roberto Calandra, editors, *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 3837–3847, Online, 26–28 Aug 2020. PMLR.

Appendix A

PUBLICATIONS INCLUDED IN THIS THESIS

Chapter 3: **D. Mouris** and N.G. Tsoutsos, "Zilch: A Framework for Deploying Transparent Zero-Knowledge Proofs." *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3269-3284, 2021. [184]

Chapter 4: **D. Mouris** and N.G. Tsoutsos, "Pythia: Intellectual Property Verification in Zero-Knowledge." *57th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, pp. 1-6, 2020. [182]

Chapter 4: **D. Mouris**, C. Gouert, and N.G. Tsoutsos, "Privacy-preserving IP verification." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 7, pp 2010–2023, 2021. [178]

Chapter 5: **D. Mouris**, C. Gouert, and N.G. Tsoutsos, "zk-Sherlock: Exposing Hardware Trojans in Zero-Knowledge." *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Nicosia, Cyprus, pp. 170-175, 2022. [179]

Chapter 6: **D. Mouris** and N.G. Tsoutsos, "Masquerade: Verifiable Multi-Party Aggregation with Secure Multiplicative Commitments." *Cryptology ePrint Archive, Report 2021/1370*, 2021. [183]

Chapter 7: **D. Mouris**, P. Sarkar, and N.G. Tsoutsos, "PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries." *Cryptology ePrint Archive, Report 2023/080*, 2023. [181]

Appendix B

PERMISSIONS

Chapter 3: The contents of this chapter have been published in [184].

Permission: © 2021 IEEE. Reprinted, with permission, from Dimitris Mouris and Nektarios G. Tsoutsos, "Zilch: A Framework for Deploying Transparent Zero-Knowledge Proofs," *IEEE Transactions on Information Forensics and Security*, 22 April 2021.

Chapter 4: The contents of this chapter have been published in [182].

Permission: © 2020 IEEE. Reprinted, with permission, from Dimitris Mouris and Nektarios G. Tsoutsos, "Pythia: Intellectual Property Verification in Zero-Knowledge," *57th ACM/IEEE Design Automation Conference (DAC)*, 20-24 July 2020.

Chapter 4: The contents of this chapter have been published in [178].

Permission: © 2021 IEEE. Reprinted, with permission, from Dimitris Mouris and Nektarios G. Tsoutsos, "Privacy-Preserving IP Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24 August 2021.

Chapter 5: The contents of this chapter have been published in [179].

Permission: © 2022 IEEE. Reprinted, with permission, from Dimitris Mouris and Nektarios G. Tsoutsos, "zk-Sherlock: Exposing Hardware Trojans in Zero-Knowledge," *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 04-06 July 2022.

Chapter 6: The contents of this chapter have been published in [183].

Permission: © 2021. Reprinted, with permission, from Dimitris Mouris and Nektarios G. Tsoutsos, "Masquerade: Verifiable Multi-Party Aggregation with Secure Multiplicative Commitments," *Cryptology ePrint Archive*.

Chapter 7: The contents of this chapter have been published in [181].

Permission: © 2023. Reprinted, with permission, from Dimitris Mouris, Pratik Sarkar, and Nektarios G. Tsoutsos, "PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries," *Cryptology ePrint Archive*.

Appendix C

ADDITIONAL PUBLICATIONS

- C. Gouert, **D. Mouris** and N.G. Tsoutsos, "SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks." *Proceedings on Privacy Enhancing Technologies (PoPETS), vol. 2023, issue. 3, pp. 154-172,* 2023

- **D. Mouris**, C. Gouert and N.G. Tsoutsos, "MPℓ $\circ$ C: Privacy-Preserving IP Verification Using Logic Locking and Secure Multiparty Computation." *IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS),* 2023

- C. Gouert, **D. Mouris** and N.G. Tsoutsos, "HELM: Navigating Homomorphic Encryption through Gates and Lookup Tables." *Cryptology ePrint Archive, Report 2023/1382,* 2023

- **D. Mouris** and N.G. Tsoutsos, "NFTs For 3D Models: Sustaining Ownership In Industry 4.0." *In IEEE Consumer Electronics Magazine,* 2022

- **D. Mouris**, C. Gouert, N. Gupta and N.G. Tsoutsos, "Peak Your Frequency: Advanced Search of 3D CAD Files in the Fourier Domain." *In IEEE Access, vol. 8, pp. 141481-141496,* 2020

- P. Cronin, C. Gouert, **D. Mouris**, N.G. Tsoutsos, and C. Yang, "Covert Data Exfiltration Using Light and Power Channels." *IEEE 37th International Conference on Computer Design (ICCD),* 2019