UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

GSKIT: A DOMAIN SPECIFIC LANGUAGE AND FRAMEWORK FOR GROTH-SAHAI PROOFS

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

BENJAMÍN IGNACIO HURTADO TORREALBA

PROFESOR GUÍA:
MATÍAS TORO IPINZA
PROFESOR GUÍA 2:
FEDERICO OLMEDO BERÓN

MIEMBROS DE LA COMISIÓN:
FELIPE BRAVO MÁRQUEZ
VALENTÍN MUÑOZ APABLAZA
FABIÁN MOSSO CHÁVEZ

SANTIAGO DE CHILE
2026

Resumen

Presentamos GSkit, un lenguaje de dominio específico (DSL) implementado en Python para la construcción de pruebas de Groth-Sahai. Las pruebas de Groth-Sahai son pruebas de conocimiento cero complejas, utilizadas en diversas aplicaciones criptográficas, cuya construcción manual es laboriosa, de bajo nivel y propensa a errores, ya que queda fuertemente acoplada a las primitivas de emparejamientos bilineales subyacentes. GSkit provee un marco de trabajo para definir estas pruebas con una probabilidad mínima de error.

El DSL permite al desarrollador especificar las variables, constantes y ecuaciones de la prueba de manera declarativa, abstrayendo la complejidad matemática. Mediante una gramática estructurada y verificación de tipos, GSkit garantiza que las pruebas estén bien formadas y sean válidas antes de ser construidas, trasladando una clase de errores desde el tiempo de ejecución al tiempo de especificación. A partir de una única especificación, el marco genera automáticamente tanto el código de generación de la prueba como el de verificación.

Evaluamos el marco mediante dos casos de estudio de complejidad creciente: el esquema de firmas BLS y el esquema de firmas basadas en atributos con enlazabilidad controlada por el usuario (ABS-UCL). Este último, al estar compuesto por varios subprotocolos que interactúan entre sí, permite comparar la implementación modular obtenida con GSkit frente a una implementación manual previa del mismo esquema construida directamente sobre una biblioteca de bajo nivel. La comparación muestra que el enfoque modular elimina la construcción de ecuaciones duplicada y mantenida a mano que caracteriza a las implementaciones manuales, reemplazándola por una única descripción declarativa de cada subprotocolo. De este modo, GSkit hace que las pruebas de Groth-Sahai sean más accesibles para los desarrolladores, reduciendo el esfuerzo y la superficie de error de su implementación sin comprometer sus garantías de seguridad criptográfica.

Abstract

We introduce GSkit, a Domain Specific Language (DSL) implemented in Python for constructing Groth-Sahai proofs. Groth-Sahai proofs are complex zero-knowledge proofs used in a wide range of cryptographic applications, whose manual construction is laborious, low-level, and error-prone, since it is tightly coupled to the underlying bilinear pairing primitives. GSkit provides a framework to define these proofs with a minimal probability of error.

The DSL allows developers to specify the variables, constants, and equations of the proof in a declarative way, abstracting away the mathematical complexity. Through a structured grammar and type checking, GSkit ensures that proofs are well-formed and valid before they are constructed, moving a class of errors from run time to specification time. From a single specification, the framework automatically generates both the proof-generation code and the verification code.

We evaluate the framework on two case studies of increasing complexity: the BLS signature scheme and the attribute-based signature scheme with user-controlled linkability (ABS-UCL). The latter, being composed of several interacting sub-protocols, allows us to compare the modular implementation obtained with GSkit against a prior manual implementation of the same scheme built directly on a low-level library. The comparison shows that the modular approach eliminates the duplicated, hand-maintained equation construction that characterizes manual implementations, replacing it with a single declarative description of each sub-protocol. In this way, GSkit makes Groth-Sahai proofs more accessible to developers, reducing the effort and error surface of their implementation without compromising their cryptographic security guarantees.

A Pascual, mi perro flaco:
el quiltro que me acompañó incondicionalmente hasta
el último de sus días.

Agradecimientos

A todos quienes me vieron en el abismo y me tendieron una mano para vivir.

Declaración de uso de inteligencia artificial

En la elaboración de esta tesis se utilizaron herramientas de inteligencia artificial generativa. Específicamente, se empleó Claude (Anthropic) en las siguientes etapas y con los siguientes propósitos: asistencia en la redacción y edición del texto escrito; apoyo como asistente de programación durante el desarrollo del código del prototipo presentado; y exploración y búsqueda de bibliografía relacionada con el tema de investigación. Todo el contenido fue revisado y validado por el autor, quien asume plena responsabilidad sobre el trabajo presentado.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem 4

1.3 Hypothesis 4

1.4 Contribution 4

2 Background **7**

2.1 Cryptography 7

2.1.1 Computational Security 8

2.2 Pairing-based Cryptography 8

2.3 Digital Signatures 9

2.4 Zero-Knowledge Proofs 9

2.4.1 Non-interactive protocol 9

2.4.2 Non-interactive Witness Indistinguishable Proofs (NIWIs) 10

2.5 Groth-Sahai Proofs 10

3 Related Work **12**

3.1 Zero-Knowledge Proof DSLs and Circuit Compilers 12

3.1.1 Circom 12

3.1.2 ZoKrates 13

3.1.3 Noir 13

3.1.4 Cairo 13

3.1.5 Leo . . . 13
3.1.6 Summary of Circuit-Based DSLs . . . 13
3.2 Groth-Sahai Proof Implementations . . . 14
3.2.1 groth-sahai-rs (Rust) . . . 14
3.2.2 groth-sahai-python (Python) . . . 14
3.2.3 groth-sahai (Java) . . . 15
3.2.4 Gap in Existing Implementations . . . 15
3.3 Formal Verification Tools for Cryptography . . . 15
3.3.1 EasyCrypt . . . 15
3.3.2 CertiCrypt and SSProve . . . 16
3.3.3 Hacspec . . . 16
3.4 Pairing-Based Cryptography Libraries . . . 16
3.4.1 Charm-Crypto . . . 16
3.4.2 arkworks . . . 16
3.4.3 PBC and RELIC . . . 17
3.5 Interoperability Standards . . . 17
3.5.1 zkInterface . . . 17
3.6 Positioning GSKit . . . 17

4 DSL for Groth-Sahai proofs 19

4.1 Grammar . . . 20
4.2 Typing Rules . . . 20
4.2.1 Variable Declaration Typing . . . 21
4.2.2 Constant Declaration Typing . . . 21
4.2.3 Equation Typing . . . 22
4.2.4 Variable extraction . . . 23
4.2.5 Typecheck . . . 24

5 Groth-Sahai Framework **29**

5.1 Eye-ball correspondence 29

5.1.1 Setup Algorithm 29

5.1.2 Soundness String Generation Algorithm 30

5.1.3 Witness-indistinguishability String Generation Algorithm 30

5.1.4 Proof Algorithm 31

5.1.5 Verify Algorithm 36

5.2 Python Decorator 37

5.3 Running the Framework 38

5.3.1 Command-Line Interface 39

5.3.2 Compilation Process 40

5.4 Evaluation 42

6 Study Cases **44**

6.1 BLS 44

6.2 ABS-UCL 45

6.2.1 Scheme Overview 45

6.2.2 Modular Construction 46

6.2.3 Code Comparison 47

6.2.4 Summary 48

7 Conclusions **49**

7.1 Future Work 50

Bibliography **53**

List of Tables

3.1 Comparison of zero-knowledge proof DSLs 14

6.1 Total developer-written lines of code (blank lines and comments excluded). . 48

6.2 Groth-Sahai equation-construction effort. The manual approach expresses every equation twice—once for proving and once for verification—whereas the modular approach derives both from a single declarative source. 48

List of Figures

1.1 Basic structure of a Zero-Knowledge Proof 2

1.2 Overview of the DSL framework workflow: from input files (‘.gs‘ and ‘.json‘) through parsing and type checking, to automatic generation of proof and verification code. The framework handles the complete pipeline from high-level specification to executable Python code for Groth-Sahai proofs. 5

5.1 Setup algorithm comparation. 29

5.2 Soundness string algorithm comparation. 30

5.3 Witness-indistinguishability algorithm comparation. 30

5.4 Proof algorithm code (part 1). 31

5.5 Proof algorithm description (part 1). 31

5.6 Proof algorithm code (part 2). 32

5.7 Proof algorithm description (part 2). 32

5.8 Proof algorithm code (part 3). 33

5.9 Proof algorithm (part 3). 33

5.10 Proof algorithm code (part 4). 34

5.11 Proof algorithm (part 4). 34

5.12 Proof algorithm code (part 5). 35

5.13 Proof algorithm (part 5). 35

5.14 Verify algorithm code. 36

5.15 Verify algorithm description. 37

5.16 BLS signature verification function decorated with `@gsfy`. 38

5.17 Proof compiler graph. 41

5.18 Verify compiler graph. 41

5.19 Performance varying number of equations 42

5.20 Performance varying number of variables 42

6.1 Manual construction (earlier implementation): the RPSPS attribute verification built by hand with explicit equation and pairing-map objects. The identical block is written a second time in the verification routine. 46

6.2 Modular construction (GSkit): the same three equations declared once inside the RPSPS sub-protocol. The `@gsfy` decorator compiles both proving and verification code from this single specification. 47

Chapter 1

Introduction

In this chapter, we introduce the fundamental motivations and challenges that inspire this work, provide a background on the Groth-Sahai proof system [17] and its significance in cryptographic protocols, outline the specific problems addressed, and present the key contributions of this thesis.

1.1 Motivation

Cryptography plays a critical role in securing information in today's digital landscape, protecting data integrity, confidentiality, and authenticity across various applications. From securing financial transactions and personal communications to enabling complex systems such as voting and identity verification, cryptographic protocols provide the foundation for trusted interactions in untrusted environments. At the heart of these protocols are mathematical structures that allow secure operations on encrypted data, enabling systems where sensitive information can be processed without exposing it to unauthorized parties.

One of the prominent advancements in cryptographic mathematical structures involves bilinear pairings, particularly on elliptic curves, which have empowered a new class of cryptographic schemes. Bilinear pairings allow for complex mathematical operations, enabling the construction of advanced protocols like non-interactive zero-knowledge (NIZK) proofs and attribute-based encryption. Such pairings allow multiple cryptographic operations to interact, making them especially useful in applications that require verifiable computations and privacy-preserving proofs.

Zero-knowledge proofs are a cryptographic technique that enables one party, the prover, to convince another party, the verifier, of the truth of a statement without revealing any information beyond the fact that the statement is true. These proofs ensure privacy by keeping sensitive data concealed while still allowing verification. Related to this concept, non-interactive witness indistinguishability (NIWI) provides a different privacy guarantee: NIWI ensures that a verifier cannot distinguish which witness was used to construct a proof among multiple valid witnesses, adding a layer of anonymity. While zero-knowledge is a

stronger property that implies witness indistinguishability, NIWI proofs can be more efficient and are valuable in scenarios where multiple valid witnesses exist.

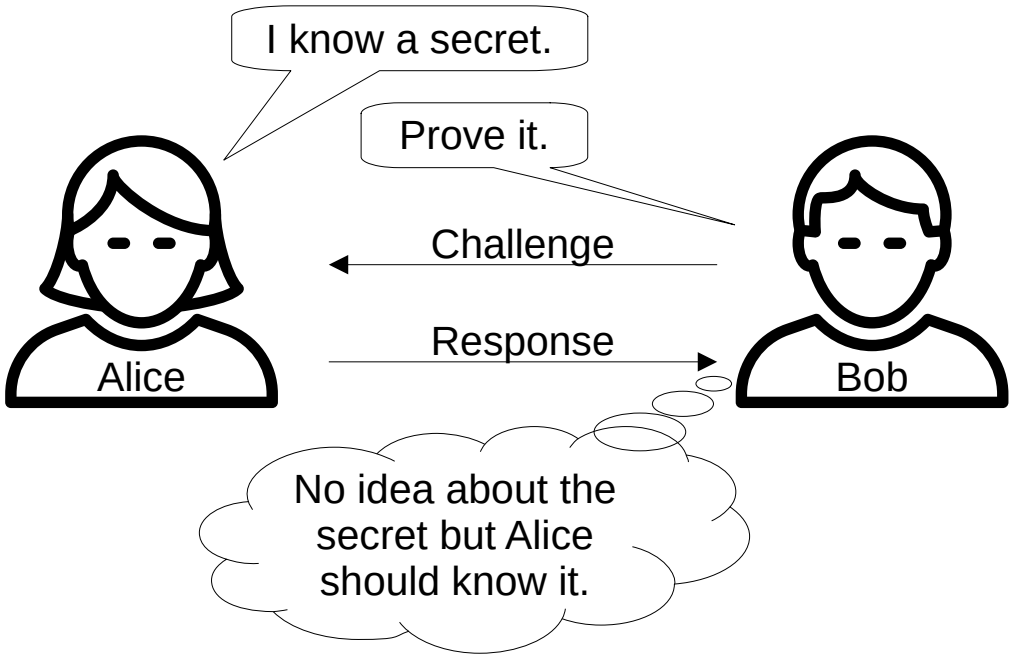

Figure 1.1: Basic structure of a Zero-Knowledge Proof

The Groth-Sahai proof system is a notable application of bilinear pairings, enabling efficient and versatile non-interactive proofs for a range of cryptographic protocols. Introduced by Jens Groth and Amit Sahai, this system offers a dual-mode operation: depending on the common reference string used, it can provide either Non-Interactive Zero-Knowledge (NIZK) or Non-Interactive Witness Indistinguishable (NIWI) proofs. This flexibility allows the system to construct proofs that verify complex statements about encrypted data while providing different privacy guarantees based on the application's requirements. Groth-Sahai proofs have gained prominence due to their efficiency and flexibility, allowing cryptographic protocols to achieve strong security guarantees with minimal computational overhead. As a result, they have become valuable tools in privacy-sensitive applications, including digital credentials, anonymous communication networks, and identity management.

Despite their potential, however, constructing Groth-Sahai proofs remains challenging. These proofs require careful attention to the underlying cryptographic primitives, bilinear group settings, and the structure of proof statements. For many researchers and developers, this complexity presents a barrier, as the current methods for building Groth-Sahai proofs demand deep expertise and are highly manual. Errors in constructing proofs can lead to implementation flaws, affecting both efficiency and security.

This complexity and the lack of high-level tooling have limited the broader adoption of Groth-Sahai proofs in real-world systems. Therefore, the development of a Domain-Specific Language (DSL) tailored to Groth-Sahai proofs could address these challenges by providing a simplified interface for proof construction. Such a DSL would abstract the intricate details, enabling cryptographers to work with high-level constructs that intuitively express their requirements, without compromising on efficiency or correctness.

The potential applications of Groth-Sahai proofs extend beyond traditional cryptographic protocols into practical real-world scenarios where privacy and authentication must coexist. A particularly compelling use case emerges in the realm of online communication and digital forums, where privacy and authentication present seemingly conflicting requirements. Users often need to maintain anonymity while simultaneously proving they possess certain attributes or credentials that qualify them to participate in specific discussions. This challenge is particularly relevant in sensitive forums where participants need to verify their expertise, role, or authority without revealing their identity.

Attribute-Based Signatures (ABS) [20] offer an elegant solution to this dilemma by leveraging the power of zero-knowledge proofs and bilinear pairings. ABS allows users to sign messages while proving they possess certain attributes, without revealing which specific attributes were used or their identity. For instance, in a medical discussion forum, a doctor could prove they have valid medical credentials without exposing their personal information or specific specialization. This capability is crucial for fostering open dialogue in professional communities while maintaining privacy and preventing potential harassment or bias based on personal information.

The implementation of such attribute-based signature schemes often relies on complex cryptographic constructions, where Groth-Sahai proofs play a vital role. These proofs enable the verification of attribute credentials and signature validity while maintaining the non-interactive and zero-knowledge properties essential for practical anonymous forum applications. However, the intricate nature of implementing these proofs correctly within an ABS system further emphasizes the need for high-level tools and abstractions that can simplify their construction and integration, reinforcing the importance of developing more accessible frameworks for working with Groth-Sahai proofs.

Building upon this concept, Attribute-Based Signatures with User-Controlled Linkability (ABS-UCL) [14][15] represent an advanced extension that provides users with even greater control over their privacy. In ABS-UCL schemes, users not only prove possession of attributes but also maintain control over the linkability of their signatures. This means users can choose whether different signatures they create should be linkable to each other, providing a balanced approach between privacy and accountability.

For instance, in a medical forum discussion, a doctor might want to maintain consistent identity across a single thread while remaining unlinkable across different discussions. ABS-UCL enables this by allowing the signer to include a linking token that they control. When the same linking token is used, signatures become linkable, but only if the signer chooses to do so. This feature is particularly valuable in scenarios where temporary consistency of identity is desired without compromising long-term privacy.

The implementation of ABS-UCL heavily relies on Groth-Sahai proofs [15] to demonstrate the validity of attributes and linking tokens while maintaining zero-knowledge properties. These proofs ensure that verifiers can confirm the authenticity of signatures and check linkability when authorized, without learning the signer's identity or specific attributes. However, the complexity of correctly implementing these proofs within an ABS-UCL system further emphasizes the need for sophisticated tools that can simplify their construction and integration.

1.2 Problem

The Groth-Sahai proof system is an efficient and versatile tool for creating non-interactive zero-knowledge (NIZK) proofs in bilinear groups. Despite its utility in cryptographic protocols, constructing these proofs requires significant expertise and manual effort. Researchers and developers often struggle with writing correct and efficient proof statements, which increases the risk of errors and implementation inefficiencies.

Currently, there is no user-friendly, high-level tool that abstracts the complexities of creating Groth-Sahai proofs, forcing cryptographers to write proofs at a low level that is tightly coupled with the underlying cryptographic primitives. This situation poses a significant barrier for adoption in real-world cryptographic systems.

Thus, there is a need for a DSL (Domain-Specific Language) that simplifies the process of constructing Groth-Sahai proofs, enabling developers to express these proofs more intuitively while maintaining efficiency and correctness.

This need becomes particularly evident in complex cryptographic schemes such as Attribute-Based Signatures with User-Controlled Linkability (ABS-UCL). These schemes require multiple intricate Groth-Sahai proofs to demonstrate attribute possession, signature validity, and controlled linkability, while maintaining zero-knowledge properties. The implementation of such proofs in ABS-UCL systems is currently error-prone and time-consuming, requiring developers to manually handle complex mathematical relationships and cryptographic commitments. A high-level DSL would significantly reduce the complexity of implementing these proofs, making advanced privacy-preserving systems more accessible to developers and researchers.

1.3 Hypothesis

A Domain-Specific Language (DSL) specifically designed for constructing Groth-Sahai proofs can significantly reduce the complexity and time required for developing non-interactive zero-knowledge (NIZK) proofs in cryptographic protocols. By abstracting the low-level details of proof construction, this DSL will enable developers and cryptographers to create accurate and efficient Groth-Sahai proofs without requiring extensive expertise in the underlying cryptographic primitives. This, in turn, will facilitate broader adoption of Groth-Sahai proofs in practical applications by lowering the barrier to entry and reducing the potential for errors in implementation.

1.4 Contribution

The main contributions of this work are:

1. A Domain-Specific Language (DSL) designed specifically for constructing general-purpose

Groth-Sahai proofs, featuring:

- An intuitive and high-level syntax for expressing proof statements
- Automatic handling of commitment schemes and pairing operations
- Type-safety and validation mechanisms to prevent common errors

2. A comprehensive framework that facilitates the integration of Groth-Sahai proofs into existing cryptographic protocols, providing:
   - A modular architecture for easy integration with existing systems
   - An extensible API for defining custom proof statements
   - Tools for proof generation and verification
3. A practical implementation and evaluation of the DSL through:
   - Implementation of real-world use cases, including BLS and ABS-UCL
   - Performance benchmarks comparing manual implementations versus DSL-based solutions
   - Analysis of the reduction in implementation complexity and potential error sources

These contributions collectively address the challenges in implementing Groth-Sahai proofs by providing a more accessible and reliable approach to proof construction while maintaining the efficiency and security properties of the underlying cryptographic primitives.

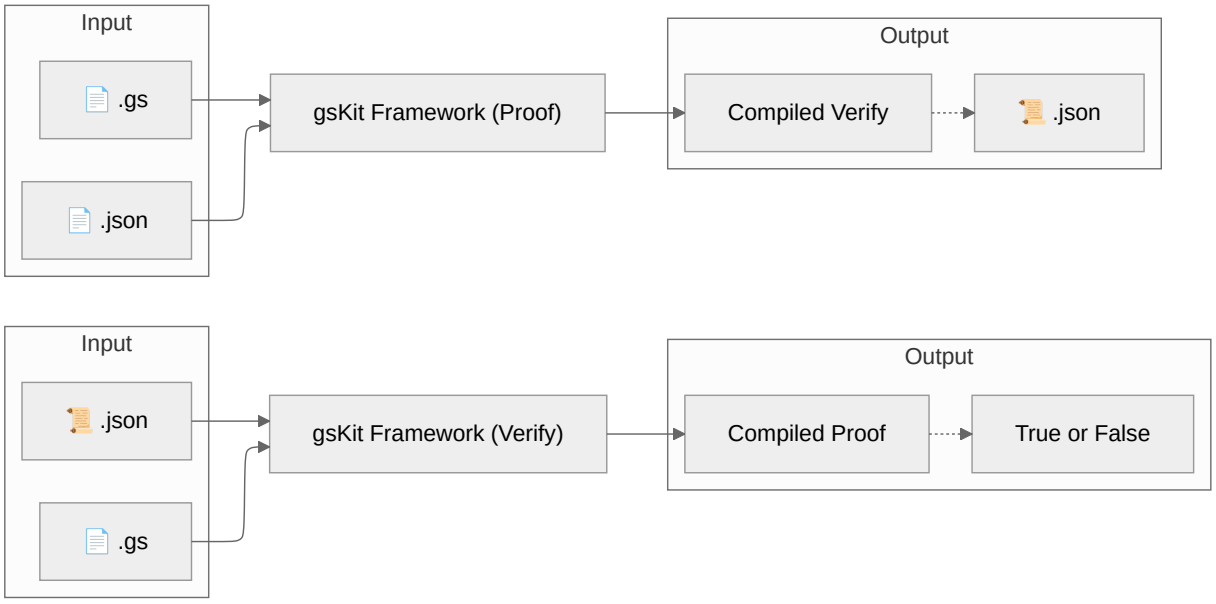

Figure 1.2: Overview of the DSL framework workflow: from input files (‘.gs‘ and ‘.json‘) through parsing and type checking, to automatic generation of proof and verification code. The framework handles the complete pipeline from high-level specification to executable Python code for Groth-Sahai proofs.

Figure 1.2 illustrates the overall architecture and workflow of our DSL framework. The system takes as input high-level specification files (‘.gs‘ files containing proof equations and ‘.json‘ files with CRS, constants, and witness definitions). The parser processes these inputs and performs type checking and validation, generating the necessary mathematical structures for proof construction. The framework then automatically generates Python code for both proof generation and verification, handling all the low-level cryptographic details such as commitment schemes, pairing operations, and proof elements. This end-to-end automation significantly reduces the complexity and potential for errors compared to manual implementation.

Chapter 2

Background

The field of cryptography has become foundational to modern secure communication systems, enabling trust and privacy in a world increasingly driven by digital interactions. This chapter provides an overview of essential cryptographic concepts and techniques that serve as building blocks for the advanced mechanisms discussed later in this thesis. Beginning with the fundamental principles of cryptography and its evolution into a computational science, the discussion progresses to pairing-based cryptography, a critical component in constructing sophisticated cryptographic protocols. Additionally, key concepts such as digital signatures, zero-knowledge proofs, and Groth-Sahai frameworks are introduced, offering a comprehensive understanding of the theoretical underpinnings essential for the development and application of the proposed framework. This background establishes a clear context for the reader, bridging foundational knowledge and specialized topics explored in subsequent chapters.

2.1 Cryptography

Cryptography, historically, was the art of concealing information by transforming it into an unintelligible format that could only be understood by those possessing a specific "secret key." This practice allowed sensitive information to be hidden from adversaries, preserving its confidentiality even if intercepted.

Modern cryptography has evolved into a rigorous scientific discipline focused on addressing a broad range of information security challenges. At its core, cryptography provides solutions to several fundamental objectives, like **Confidentiality** (Ensuring that information remains private and accessible only to authorized parties), **Integrity** (Verifying that data has not been tampered with or altered during transmission or storage), **Authenticity** (Confirming the identity of the sender or recipient of a message, ensuring that they are who they claim to be), **Non-repudiation** (Preventing parties from denying their involvement in a transaction or communication), etc.

2.1.1 Computational Security

The security of a cryptographic design relies on the system should be, if not mathematically, indecipherable. In other words, Security is only preserved against *efficient* adversaries and adversaries can potentially succeed with some *very small probability*.

Using a asymptotic approach, we say that an *efficient adversary* is an algorithm that runs in polynomial time in the security parameter λ ($a \cdot c^{\lambda}$) and *small probability of success* is a smaller probability than any inverse-polynomial in λ (λ^{-c}). A function that grows slower than any inverse polynomial is called *negligible*.

2.2 Pairing-based Cryptography

Definition 2.1 *Consider three groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order p. Let g_1, g_2 generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. A* bilinear pairing *is a map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ such that:*

- *Bilinearity:* $\forall a, b \in \mathbb{Z}_p : e\left(\lfloor a \rfloor g_1, \lfloor b \rfloor g_2\right) = e\left(g_1, g_2\right)^{ab}$
- *Non-Degeneracy:* $e\left(g_1, g_2\right) \neq 1_{\mathbb{G}_T}$
- *For practical purposes, e has to be efficiently computable.*

Usually, the group that is used for bilinear pairing are based in Elliptic Curves.

There are three classes of bilinear maps:

1. $\mathbb{G}_1 = \mathbb{G}_2$ (*symmetric pairing*)
2. $\mathbb{G}_1 \neq \mathbb{G}_2$ but there is an efficiently computable homomorphism $\Phi : \mathbb{G}_2 \to \mathbb{G}_1$
3. $\mathbb{G}_1 \neq \mathbb{G}_2$ and there are no efficiently computable homomorphisms between $\mathbb{G}_2$ and $\mathbb{G}_1$

In what follows, we will use the third class of bilinear maps, particularly the BN254 curve implemented in [2].

The possible operations with additive notation ($\mathbb{Z}_p$, $\mathbb{G}_1$ and $\mathbb{G}_2$) and multiplicative notation ($\mathbb{G}_T$) are as follows:

	$\mathbb{Z}_p$	$\mathbb{G}_1$	$\mathbb{G}_2$	$\mathbb{G}_T$
$\mathbb{Z}_p$	$\texttt{add} \to \mathbb{Z}_p$ $\texttt{mul} \to \mathbb{Z}_p$	$\texttt{mul} \to \mathbb{G}_1$	$\texttt{mul} \to \mathbb{G}_2$	$\perp$
$\mathbb{G}_1$	$\perp$	$\texttt{add} \to \mathbb{G}_1$	$\texttt{pair} \to \mathbb{G}_T$	$\perp$
$\mathbb{G}_2$	$\perp$	$\perp$	$\texttt{add} \to \mathbb{G}_2$	$\perp$
$\mathbb{G}_T$	$\texttt{pow} \to \mathbb{G}_T$	$\perp$	$\perp$	$\texttt{mul} \to \mathbb{G}_T$

2.3 Digital Signatures

A Digital Signature Scheme consists of three algorithms:

- $\textsc{KeyGen}(1^\lambda)$: takes as input a security parameter 1^λ and outputs for a signer, a pair of a secret signing key and public verification key (sk, vk).
- $\textsc{Sign}(m, \text{sk})$: takes as input a message m and the signer secret key sk, and outputs a signature σ.
- $\textsc{Verify}(\sigma, m, \text{vk})$: outputs 1 if signature verifies correctly and 0 otherwise.

A signature scheme that we will use extensively in this thesis is the Boneh-Lynn-Shacham (BLS) signature scheme [11]. The BLS signature scheme uses bilinear pairings to create signatures. The scheme works as follows:

- $\textsc{KeyGen}(1^\lambda)$: Choose a random secret key $\text{sk} \leftarrow \mathbb{Z}_p$ and compute the verification key as $\text{vk} = \text{sk} \cdot g_2$ where g_2 is a generator of $\mathbb{G}_2$
- $\textsc{Sign}(m, \text{sk})$: Compute $\sigma = \text{sk} \cdot \mathcal{H}(m)$ where $\mathcal{H}$ is a hash function that maps messages to elements in $\mathbb{G}_1$
- $\textsc{Verify}(\sigma, m, \text{vk})$: Check if $e(\sigma, g_2) = e(\mathcal{H}(m), \text{vk})$

The security of BLS signatures relies on the computational Diffie-Hellman assumption in bilinear groups and the random oracle model for the hash function $\mathcal{H}$. The verification equation makes use of the bilinear pairing e and forms a pairing product equation that will be important when constructing zero-knowledge proofs about BLS signatures later in this thesis.

2.4 Zero-Knowledge Proofs

Zero-Knowledge proofs are a general class of protocols between two parties, called the prover P and the verifier V, which enable P to convince V that a given statement is true without revealing any additional information beyond the validity of the statement itself.

2.4.1 Non-interactive protocol

A *non-interactive protocol* is a type of Zero-Knowledge proof where the prover P can generate a proof that can be verified by V without requiring multiple rounds of communication. This is often achieved using a common reference string (CRS), which both parties have access to and use to produce and verify proofs. Non-interactive protocols are particularly useful in

settings where communication is constrained, or in applications like blockchain, where proving authenticity without revealing private information is crucial. Notable examples include the Fiat-Shamir heuristic [16], which transforms interactive proofs into non-interactive ones under the assumption of a random oracle model.

2.4.2 Non-interactive Witness Indistinguishable Proofs (NIWIs)

Non-interactive Witness Indistinguishable Proofs (NIWIs) are a class of non-interactive proofs in which the verifier V cannot distinguish between different witnesses that the prover P might use to establish the truth of a statement. In other words, given two possible witnesses that satisfy the statement, the proof provided by P does not reveal any information about which specific witness was used. NIWIs find applications in scenarios where multiple witnesses are valid, and it is important to keep the actual witness hidden. They are also favored in privacy-preserving applications, where anonymity is paramount, as in certain cryptographic voting or anonymous credential systems.

It is important to note the relationship between Witness Indistinguishability (WI) and Zero-Knowledge (ZK): while Zero-Knowledge is a stronger property that implies Witness Indistinguishability, the converse is not true. That is, every Non-Interactive Zero-Knowledge (NIZK) proof is also a NIWI, but NIWIs do not necessarily satisfy the Zero-Knowledge property. A NIWI protocol guarantees that the verifier cannot determine which witness was used among multiple valid witnesses, but it may still leak some information about the witness itself. In contrast, a NIZK protocol ensures that no information beyond the validity of the statement is revealed, which automatically implies witness indistinguishability.

2.5 Groth-Sahai Proofs

Groth-Sahai proofs[17] are a family of non-interactive proofs that leverage bilinear group pairings to construct efficient and versatile proofs for certain equations. Groth-Sahai proofs are celebrated for their efficiency and composability, enabling a wide range of applications in cryptographic protocols. They are especially useful in protocols that require simultaneous proofs of multiple statements or proofs of statements involving conjunctions or disjunctions of variables, such as those found in Attribute-Based Signatures or Multi-party Computation (MPC) settings.

In addition to witness indistinguishability, Groth-Sahai proofs also satisfy *soundness*: no adversary, however powerful, should be able to produce a valid-looking proof for a false statement. Groth-Sahai proofs achieve both properties through a dual-mode common reference string. Under the SXDH assumption, the CRS can be sampled in a *soundness mode*, where the commitments become perfectly binding and the proof system is perfectly sound, or in a *witness-indistinguishability mode*, where the commitments become perfectly hiding and the proof system is perfectly witness-indistinguishable. The two CRS distributions are computationally indistinguishable under SXDH, which is what allows the security proof to reason about one mode while the real protocol runs in the other. Chapter 5 shows how GSkit

derives both CRS variants — the Soundness String Generation Algorithm and the Witness-indistinguishability String Generation Algorithm — side by side with the underlying Python implementation.

The equations that describes a Groth-Sahai proof are as follows:

With variables $\vec{\mathcal{X}} \in \mathbb{G}_1^m$, $\vec{\mathcal{Y}} \in \mathbb{G}_2^n$, $\vec{x} \in \mathbb{Z}_p^{m'}$ and $\vec{y} \in \mathbb{Z}_p^{n'}$

- Pairing Product Equation:

$$\prod_{i=1}^{n} e(\mathcal{A}_i, \mathcal{Y}_i) \cdot \prod_{i=1}^{m} e(\mathcal{X}_i, \mathcal{B}_i) \cdot \prod_{i=1}^{m} \prod_{j=1}^{n} e(\mathcal{X}_i, \mathcal{Y}_j)^{\gamma_{ij}} = t_T$$

  for constants $\mathcal{A}_i \in \mathbb{G}_1$, $\mathcal{B}_i \in \mathbb{G}_2$, $t_T \in \mathbb{G}_T$, $\gamma_{ij} \in \mathbb{Z}_p$

- Multi-Scalar Multiplication Equation in $\mathbb{G}_1$:

$$\sum_{i=1}^{n\prime} y_i \mathcal{A}_i + \sum_{i=1}^{m} b_i \mathcal{X}_i + \sum_{i=1}^{m} \sum_{j=1}^{n\prime} \gamma_{ij} y_j \mathcal{X}_i = \mathcal{T}_1$$

  for constants $\mathcal{A}_i, \mathcal{T}_1 \in \mathbb{G}_1$ and $b_i, \gamma_{ij} \in \mathbb{Z}_p$

- Multi-Scalar Multiplication Equation in $\mathbb{G}_2$:

$$\sum_{i=1}^{n} a_i \mathcal{Y}_i + \sum_{i=1}^{m\prime} x_i \mathcal{B}_i + \sum_{i=1}^{m\prime} \sum_{j=1}^{n} \gamma_{ij} x_i \mathcal{Y}_j = \mathcal{T}_2$$

  for constants $\mathcal{B}_i, \mathcal{T}_2 \in \mathbb{G}_2$ and $a_i, \gamma_{ij} \in \mathbb{Z}_p$

- Quadratic Equation:

$$\sum_{i=1}^{n\prime} a_i y_i + \sum_{i=1}^{m\prime} x_i b_i + \sum_{i=1}^{m\prime} \sum_{j=1}^{n\prime} \gamma_{ij} x_i y_j \equiv t \pmod{p}$$

  for constants p$a_i, b_i, \gamma_{ij}, t \in \mathbb{Z}_p$

Chapter 3

Related Work

This chapter reviews related work in zero-knowledge proof systems, domain-specific languages for cryptographic proofs, and tools for pairing-based cryptography. We position GSKit within this landscape, identifying the specific gap it addresses: a declarative DSL for Groth-Sahai proof construction with modular composition capabilities.

3.1 Zero-Knowledge Proof DSLs and Circuit Compilers

The past decade has seen significant development in domain-specific languages for zero-knowledge proofs. However, these tools predominantly target circuit-based proof systems (R1CS, PLONK, STARKs) rather than pairing-based equation systems like Groth-Sahai.

3.1.1 Circom

Circom [9] is the most widely adopted DSL for zero-knowledge circuit development, created by Jordi Baylina and the iden3 team. Circom compiles circuit descriptions into R1CS (Rank-1 Constraint System) representations, which can then be used with various proving backends including Groth16 and PLONK.

Circom has been used in prominent real-world applications such as Tornado Cash (privacy-preserving transactions) and Dark Forest (on-chain game with hidden information). Its success stems from optimized WebAssembly proof generation for browsers and efficient on-chain verification.

However, Circom operates at the constraint level, requiring developers to think in terms of arithmetic circuits. This abstraction is poorly suited for pairing-based protocols where the natural representation is bilinear equations over group elements, not arithmetic constraints over field elements.

3.1.2 ZoKrates

ZoKrates [13] was one of the first tools to provide a high-level language for zkSNARK development, presented at IEEE S&P 2018. It offers a Python-like syntax that compiles to R1CS constraints and supports multiple backends.

Like Circom, ZoKrates targets the circuit paradigm. While it provides higher-level abstractions than Circom, it still requires expressing computations as arithmetic circuits, making it unsuitable for direct specification of pairing product equations.

3.1.3 Noir

Noir [6], developed by Aztec, takes a Rust-inspired approach to zero-knowledge programming. Its key innovation is compilation to ACIR (Abstract Circuit Intermediate Representation), which can be further compiled to different backend proof systems.

Benchmarks show Noir's UltraPlonk backend achieving over 30x speedup compared to Circom's PLONK for large circuits like SHA-256 [26]. However, Noir remains circuit-focused and does not provide native support for pairing operations or bilinear group equations.

3.1.4 Cairo

Cairo [22] is the language underlying StarkNet and StarkEx, enabling STARK-proven general computation. It has powered applications including dYdX, Sorare, and Immutable X, processing significant transaction volumes on Ethereum.

Cairo 1.0 introduced mainstream language features that improved developer accessibility. However, Cairo targets STARKs, which are based on different mathematical foundations (algebraic intermediate representations) than Groth-Sahai proofs (bilinear groups and pairings).

3.1.5 Leo

Leo [3] is the primary language for the Aleo blockchain, a privacy-focused smart contract platform. It compiles to R1CS constraints for Aleo's zkSNARK system.

3.1.6 Summary of Circuit-Based DSLs

The key observation is that all mainstream ZK DSLs compile to constraint systems designed for general-purpose computation. None provide direct support for the bilinear equation systems that arise naturally in pairing-based cryptography. This forces developers implementing pairing-based protocols to either work at a very low level or translate their natural equation representations into circuit constraints, losing clarity and introducing potential errors.

Tool	Target	Proof System	Abstraction Level
Circom	R1CS	Groth16, PLONK	Low (constraints)
ZoKrates	R1CS	Groth16, GM17	Medium (programs)
Noir	ACIR	UltraPlonk, others	High (Rust-like)
Cairo	AIR	STARKs	High (programs)
Leo	R1CS	Marlin	Medium (programs)
GSKit	Bilinear equations	Groth-Sahai	High (equations)

Table 3.1: Comparison of zero-knowledge proof DSLs

3.2 Groth-Sahai Proof Implementations

Several implementations of the Groth-Sahai proof system exist, but they provide low-level APIs rather than declarative specification languages.

3.2.1 groth-sahai-rs (Rust)

The `groth-sahai-rs` library [25] provides a Rust implementation of the Groth-Sahai proof system built on the arkworks ecosystem. It supports witness-indistinguishable and zero-knowledge proofs for satisfiability of equations over bilinear groups.

Key characteristics:

- Requires the SXDH assumption and Type-III pairings
- Uses arkworks for elliptic curve arithmetic (e.g., BLS12-381)
- Provides programmatic API, not a declarative language
- Explicitly marked as academic proof-of-concept, not production-ready

While `groth-sahai-rs` offers efficient Rust implementations, developers must manually construct equation structures, manage variable/constant classifications, and handle proof composition programmatically.

3.2.2 groth-sahai-python (Python)

The `groth-sahai-python` library [24] is a reference implementation using the `py_ecc` library for elliptic curve operations. It serves primarily as an educational resource and prototype implementation.

This library demonstrates that Python can be used for Groth-Sahai proofs, but it provides only low-level functions without any abstraction for specifying proof equations declaratively.

3.2.3 groth-sahai (Java)

An experimental Java implementation [23] of the Groth-Sahai NIZK protocol exists, though it is explicitly marked as unsuitable for production use.

3.2.4 Gap in Existing Implementations

All existing Groth-Sahai implementations share a common limitation: they provide *programmatic APIs* that require developers to manually:

1. Classify variables as witnesses or public inputs
2. Determine equation types (PPE, MS1, MS2, QE)
3. Construct commitment and proof structures
4. Manage variable naming across composed proofs

This manual process is error-prone and does not scale well to complex protocols involving multiple sub-proofs, such as attribute-based signatures with policy predicates.

3.3 Formal Verification Tools for Cryptography

Several tools focus on formal verification of cryptographic protocols, providing strong security guarantees but requiring significant expertise and not directly generating executable implementations.

3.3.1 EasyCrypt

EasyCrypt [7] is an SMT-based verification framework for constructing and verifying game-based cryptographic security proofs. It uses a probabilistic programming language to express security games and adversary interactions.

EasyCrypt has been used to verify security proofs for schemes including OAEP encryption and various signature schemes. Recent work has connected EasyCrypt with the Jasmin language for verifying low-level cryptographic implementations [18].

However, EasyCrypt focuses on *proving security* of protocols rather than *generating implementations*. A cryptographer using EasyCrypt still needs to implement the protocol separately.

3.3.2 CertiCrypt and SSProve

CertiCrypt [8] embeds cryptographic verification in the Coq theorem prover, enabling machine-checked security proofs. The more recent SSProve framework [1] provides foundational verification for modular cryptographic proofs.

These tools are complementary to our work: GSKit could potentially be connected to such verification frameworks to provide formally verified proof generation, but this is left for future work.

3.3.3 Hacspec

Hacspec [10] is a specification language for cryptographic primitives designed to be readable, executable, and compilable to formal verification backends (F*, Coq, EasyCrypt).

Hacspec focuses on specifying *algorithms* (e.g., AES, ChaCha20) rather than *proof systems*. While valuable for cryptographic primitive specification, it does not address the specific challenges of zero-knowledge proof construction.

3.4 Pairing-Based Cryptography Libraries

GSKit builds upon existing pairing cryptography libraries, particularly Charm-Crypto.

3.4.1 Charm-Crypto

Charm [2] is a Python framework for rapidly prototyping advanced cryptosystems. It uses a hybrid design with performance-intensive operations in native C modules while cryptosystems are written in high-level Python.

Charm supports multiple mathematical backends including PBC, MIRACL, and RELIC, allowing performance comparison across libraries. It provides abstractions for integer groups, elliptic curve groups, and pairing groups.

GSKit uses Charm-Crypto as its default backend for pairing operations, benefiting from its mature implementation while adding a declarative layer for Groth-Sahai proof specification.

3.4.2 arkworks

The arkworks ecosystem [5] provides Rust libraries for zkSNARK development, including efficient implementations of pairing-friendly curves (BLS12-381, BN254) and finite field arithmetic.

While arkworks is primarily designed for circuit-based SNARKs, the `groth-sahai-rs` library demonstrates that it can support Groth-Sahai proofs. A future version of GSKit could target arkworks as an alternative backend for improved performance.

3.4.3 PBC and RELIC

The PBC (Pairing-Based Cryptography) library [19] from Stanford was designed specifically for pairing-based protocols and remains widely used despite not offering state-of-the-art performance.

RELIC [4] is a more recent library built for speed and portability, with optimizations for pairing operations. Charm-Crypto can use RELIC as a backend, providing a path for GSKit to benefit from these optimizations.

3.5 Interoperability Standards

3.5.1 zkInterface

zkInterface [21] is a protocol for interoperability between zero-knowledge proof frameworks, developed through the ZKProof community. It specifies formats for communicating constraint systems and variable assignments between frontends (compilers) and backends (provers/verifiers).

zkInterface focuses on R1CS-style constraint systems and does not currently support pairing-based equation systems. However, its design philosophy of separating frontend specification from backend implementation aligns with GSKit's architecture, where the DSL specification is independent of the underlying pairing library.

3.6 Positioning GSKit

GSKit addresses a specific gap in the landscape of zero-knowledge proof tools:

1. **Declarative specification**: Unlike programmatic APIs (`groth-sahai-rs`, `groth-sahai-python`) GSKit allows specifying proof equations in a declarative DSL that closely matches mathematical notation.
2. **Native pairing support**: Unlike circuit compilers (Circom, Noir, ZoKrates), GSKit directly supports bilinear equations over $\mathcal{G}_1$, $\mathcal{G}_2$, $\mathcal{G}_T$, and $\mathbb{Z}_p$ without requiring translation to arithmetic constraints.
3. **Automatic type inference**: The DSL automatically determines equation types (PPE, MS1, MS2, QE) and variable classifications based on how they appear in equations.

4. **Modular composition**: The `GSContext` mechanism enables combining multiple proof components with automatic variable renaming and conflict detection, essential for complex protocols like attribute-based signatures.

5. **Code generation**: Unlike formal verification tools (EasyCrypt, CertiCrypt), GSKit generates executable proof and verification code, bridging specification and implementation.

The key insight is that Groth-Sahai proofs remain valuable for pairing-based protocols because they provide a "natural and direct way to prove statements that arise in pairing-based cryptography, without having to first compile the statement to a circuit" [12]. GSKit makes this natural representation accessible through a high-level DSL while handling the complex details of proof construction automatically.

Chapter 4

DSL for Groth-Sahai proofs

Domain Specific Languages (DSLs) play a crucial role in modern software development by providing specialized syntax and semantics tailored to specific problem domains. In the context of cryptographic protocols, and particularly for zero-knowledge proof systems like Groth-Sahai proofs, a well-designed DSL can significantly reduce complexity and potential errors while improving readability and maintainability.

The construction of Groth-Sahai proofs involves intricate mathematical operations over bilinear groups and requires careful handling of various equation types including pairing product equations, multi-scalar multiplication equations, and quadratic equations. Without proper abstraction, implementing these proofs can be error-prone and time-consuming, even for experienced cryptographers. Moreover, verifying the correctness of such implementations becomes increasingly challenging as the complexity of the proof system grows.

Our DSL addresses these challenges by providing:

- **Type Safety**: Static typing ensures that mathematical operations are valid and consistent with the underlying algebraic structures.
- **Abstraction**: High-level representations of complex mathematical concepts that hide implementation details while preserving semantic clarity.
- **Composability**: The ability to construct complex proofs from simpler components in a modular and reusable way.
- **Verification**: Built-in validation mechanisms that catch common errors at compile-time rather than runtime.

In this chapter, we introduce a novel Domain Specific Language (DSL) designed specifically for constructing Groth-Sahai proofs, enhancing both usability and efficiency in cryptographic applications. This DSL provides a high-level abstraction that simplifies the representation and manipulation of complex mathematical structures inherent in the Groth-Sahai proof system. By incorporating intuitive syntax and robust semantic features, our DSL enables humans and machines* to articulate proof constructions with greater clarity and

fewer errors. Furthermore, the language integrates optimization techniques that facilitate the generation of more efficient proof artifacts, ultimately advancing the state of the art in zero-knowledge proof systems. Through illustrative examples and comparative analysis, we demonstrate the advantages of our DSL in streamlining the proof development process and improving overall performance.

4.1 Grammar

(Values)	v ::=	`ZpElement`	string repr.
	|	`G1Element`	string repr.
	|	`G2Element`	string repr.
	|	`GTElement`	string repr.
(Types)	τ ::=	`Zp`	$\mathbb{Z}_p$ element type
	|	`G1`	$\mathbb{G}_1$ element type
	|	`G2`	$\mathbb{G}_2$ element type
	|	`GT`	$\mathbb{G}_T$ element type
(Declaration)	d ::=	`NAME` : τ	type annotation
(MulTerm)	m ::=	`NAME` $*$ `NAME`	
	|	`NAME` $*$ `NAME` $*$ `NAME`	
(Equation)	e ::=	m $[+\ m]^*$ `=` `NAME`	
(Specs)	σ ::=	`variables:` $d_1 \dots d_n$	variables block
	|	`constants:` $d'_1 \dots d'_m$	constants block
	|	`equations:` $e_1 \dots e_k$	equations block

4.2 Typing Rules

1. We initialize the environment Γ from variable and constant declarations. For each declaration in the variables and constants blocks, we add the corresponding type to the environment. Each `NAME` must be unique (no duplicates).

$$\emptyset \vdash_{decl} \texttt{variables:} \quad decl_1 \dots decl_n \rightsquigarrow \Gamma_v$$

$$\Gamma_v \vdash_{decl} \texttt{constants:} \quad decl'_1 \dots decl'_m \rightsquigarrow \Gamma$$

2. We improve the environment by iterating over each equation and assigning for every `Var[ZP]` in the environment Γ to $\texttt{Var}_x\texttt{[ZP]}$ or $\texttt{Var}_y\texttt{[ZP]}$.

$$\Gamma \vdash_A e \rightsquigarrow \Gamma'$$

3. We distribute each `NAME` in the Γ' environment to differentiated environment for each type `Var[G1]`, `Var[G2]`, $\texttt{Var}_x\texttt{[ZP]}$, $\texttt{Var}_y\texttt{[ZP]}$ and `Const[_]`.

$$\texttt{ext}(\Gamma'; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset) \rightsquigarrow X; Y; x; y; \mathcal{C}$$

4. Finally, we elaborate on each equation by creating the vectors a and b along with a matrix γ.

$$X; Y; x; y; \mathcal{C} \vdash_e e \rightsquigarrow a; b; \gamma$$

4.2.1 Variable Declaration Typing

$$\frac{p \notin \mathrm{dom}(\Gamma) \qquad \tau \in \{\texttt{ZP}, \texttt{G1}, \texttt{G2}\}}{\Gamma \vdash_{decl} p : \tau : \Gamma[p \mapsto \texttt{Var[}\tau\texttt{]}]} \text{(New Variable)}$$

Rule New Variable type-checks a single declaration $p : \tau$: it requires p to be fresh in Γ and $\tau \in \{\texttt{ZP}, \texttt{G1}, \texttt{G2}\}$, extending the environment by mapping p to `Var[`τ`]` — an uncommitted witness variable of that type.

$$\frac{\Gamma \vdash_{decl} decl_1 : \Gamma_1 \qquad \Gamma_1 \vdash_{decl} decl_2 \ldots decl_n : \Gamma_n}{\Gamma \vdash_{decl} decl_1 \ldots decl_n : \Gamma_n} \text{(All Variables)}$$

Rule All Variables sequences a list of declarations by threading the environment left-to-right: $decl_1$ is checked in Γ yielding Γ_1, then the remaining $decl_2 \ldots decl_n$ are checked in Γ_1, accumulating into Γ_n.

4.2.2 Constant Declaration Typing

$$\frac{c \notin \mathrm{dom}(\Gamma) \qquad \tau \in \{\texttt{ZP}, \texttt{G1}, \texttt{G2}, \texttt{GT}\}}{\Gamma \vdash_{decl} c : \tau : \Gamma[c \mapsto \texttt{Const[}\tau\texttt{]}]} \text{(New Constant)}$$

Rule New Constant mirrors New Variable: a fresh name c with type $\tau \in \{\texttt{ZP}, \texttt{G1}, \texttt{G2}, \texttt{GT}\}$ is added to Γ as `Const[`τ`]`, distinguishing it from witness variables. Note that `GT` is permitted here but not for variables.

$$\frac{\Gamma \vdash_{decl} decl'_1 : \Gamma_1 \qquad \Gamma_1 \vdash_{decl} decl'_2 \ldots decl'_m : \Gamma_m}{\Gamma \vdash_{decl} decl'_1 \ldots decl'_m : \Gamma_m} \text{(All Constants)}$$

Rule All Constants sequences a list of constant declarations in the same left-to-right threading as All Variables, producing the final environment Γ_m.

4.2.3 Equation Typing

These rules propagate the role of $\mathbb{Z}_p$ variables — x-side or y-side — based on the position they occupy within products.

$$\frac{\Gamma(p) = \texttt{Var}_d\texttt{[ZP]} \quad d \in \{x, \bot\}}{\Gamma \vdash_x p : \Gamma[p \mapsto \texttt{Var}_x\texttt{[ZP]}]} \text{(Tp)}$$

Rule Tp promotes p to $\texttt{Var}_x\texttt{[ZP]}$ when it appears on the left of a product and currently has an undecided or x-compatible $\mathbb{Z}_p$ type.

$$\frac{\Gamma(p) \neq \texttt{Var}_d\texttt{[ZP]}}{\Gamma \vdash_x p : \Gamma} \text{(Tp-Skip)}$$

Rule Tp-Skip leaves Γ unchanged when p is not of that kind.

$$\frac{\Gamma(q) \neq \texttt{Var}_d\texttt{[ZP]}}{\Gamma \vdash_y q : \Gamma} \text{(Tq-Skip)}$$

The symmetric rule Tq-Skip handles the right-hand factor q analogously when q is not a $\mathbb{Z}_p$ variable.

$$\frac{\Gamma \vdash_x p : \Gamma_p \quad \Gamma_p \vdash_y q : \Gamma_q}{\Gamma \vdash_B p * q : \Gamma_q} \text{(Pair)}$$

Rule Pair composes both: it applies $\vdash_x$ to p and $\vdash_y$ to q in sequence, classifying each factor in a single multiplication term.

$$\frac{\Gamma \vdash_B p * q : \Gamma'}{\Gamma \vdash_A [r*]p * q : \Gamma'} \text{(Single)}$$

At the equation level, Single lifts a single checked term $[r*]p * q$ into the equation judgment.

(Extend)

$$\frac{\Gamma \vdash_A \sum_{i=0}^{l} [r_i *] p_i * q_i : \Gamma' \quad \Gamma' \vdash_B p * q : \Gamma''}{\Gamma \vdash_A \sum_{i=0}^{l} [r_i *] p_i * q_i + [r *] p * q : \Gamma''}$$

Extend folds each subsequent term into the accumulated environment by threading it through Pair.

4.2.4 Variable extraction

The function `ext` partitions the refined environment Γ' into five disjoint collections: X collects $\mathbb{G}_1$ variables, Y collects $\mathbb{G}_2$ variables, x collects $\texttt{Var}_x\texttt{[ZP]}$ scalars, y collects $\texttt{Var}_y\texttt{[ZP]}$ scalars, and $\mathcal{C}$ collects all constants. Its signature is:

$$\texttt{ext} : \Gamma; X; Y; x; y; \mathcal{C} \to X; Y; x; y; \mathcal{C}$$

$$\texttt{ext}\,(\emptyset; X; Y; x; y; \Gamma') \rightsquigarrow X; Y; x; y; \mathcal{C}$$

The base case, when Γ is empty, returns the five accumulated sets.

$$\texttt{ext}\,(\Gamma, p : \texttt{Var[G1]}; X; Y; x; y; \mathcal{C}) \rightsquigarrow \texttt{ext}\,(\Gamma; X, p; Y; x; y; \mathcal{C})$$

This case consumes a $\mathbb{G}_1$ binding p from Γ, appends it to X, and recurses.

$$\texttt{ext}\,(\Gamma, q : \texttt{Var[G2]}; X; Y; x; y; \mathcal{C}) \rightsquigarrow \texttt{ext}\,(\Gamma; X; Y, q; x; y; \mathcal{C})$$

This case consumes a $\mathbb{G}_2$ binding q from Γ, appends it to Y, and recurses.

$$\texttt{ext}\,(\Gamma, p : \texttt{Var}_x\texttt{[ZP]}; X; Y; x; y; \mathcal{C}) \rightsquigarrow \texttt{ext}\,(\Gamma; X; Y; x, p; y; \mathcal{C})$$

This case consumes a $\texttt{Var}_x\texttt{[ZP]}$ binding p from Γ, appends it to x, and recurses.

$$\texttt{ext}\,(\Gamma, q : \texttt{Var}_y\texttt{[ZP]}; X; Y; x; y; \mathcal{C}) \rightsquigarrow \texttt{ext}\,(\Gamma; X; Y; x; y, q; \mathcal{C})$$

This case consumes a $\texttt{Var}_y\texttt{[ZP]}$ binding q from Γ, appends it to y, and recurses.

$$\texttt{ext}\,(\Gamma, c : \texttt{Const[\_]}; X; Y; x; y; \mathcal{C}) \rightsquigarrow \texttt{ext}\,(\Gamma; X; Y; x; y; \mathcal{C}, c)$$

This case consumes a constant binding c from Γ, appends it to $\mathcal{C}$, and recurses.

4.2.5 Typecheck

Before presenting the typing rules for equations, we introduce two auxiliary operations used throughout this section. The merge operator $\curlyvee$ combines two sparse structures — vectors or matrices whose entries are each either $\perp$ (undetermined) or a concrete value — by taking whichever side is non-$\perp$ at each position, and failing if both sides are non-$\perp$ and disagree at the same position; this is how the *-EQ rules below combine the independently-typed contribution of each term into a single equation. Formally:

$$\begin{aligned} a \curlyvee \perp &:= a \\ \perp \curlyvee a &:= a \\ a' \curlyvee a &:= \texttt{Error} \end{aligned}$$

The index function $\texttt{gi}(p; x)$ returns the 1-indexed position of parameter p within the sequence x, or -1 if p does not occur in x; it is used to place a variable's coefficient at the correct position of a sparse vector or matrix. Formally, by recursion on x:

$$\begin{aligned} \texttt{gi}(p; x, q) &:= \texttt{gi}(p; x) \\ \texttt{gi}(p; x, p) &:= \texttt{len}(x) + 1 \\ \texttt{gi}(p; \emptyset) &:= -1 \end{aligned}$$

Quadratic Equation

These four rules typecheck multiplication terms and equations where all variables live in $\mathbb{Z}_p$.

(QE-LEFT)

$$\frac{X; Y; x; y; \mathcal{C} \vdash p : \texttt{Const[ZP]} \qquad X; Y; x; y; \mathcal{C} \vdash q : \texttt{Var}_y\texttt{[ZP]} \qquad a = \left\{ \begin{array}{ll} \perp & \text{if } j \neq \texttt{gi}(q; y), \\ p & \text{if } j = \texttt{gi}(q; y) \end{array} \right\}_{j=1}^{|y|} \qquad b = \{\perp\}^{|x|} \qquad \gamma = \{\perp\}^{|x| \times |y|}}{X; Y; x; y; \mathcal{C} \vdash_m p * q : \texttt{Mul[QE]} \rightsquigarrow a; b; \gamma}$$

Rule QE-LEFT handles a $\mathbb{Z}_p$ constant p multiplied by an x-variable q: a is a sparse vector with value p at q's index, and b, γ are all $\perp$.

(QE-RIGHT)

$$\frac{X; Y; x; y; \mathcal{C} \vdash p : \texttt{Var}_x\texttt{[ZP]} \qquad X; Y; x; y; \mathcal{C} \vdash q : \texttt{Const[ZP]} \qquad a = \{\perp\}^{|y|} \qquad b = \left\{ \begin{array}{ll} \perp & \text{if } j \neq \texttt{gi}(p; x), \\ q & \text{if } j = \texttt{gi}(p; x) \end{array} \right\}_{j=1}^{|x|} \qquad \gamma = \{\perp\}^{|x| \times |y|}}{X; Y; x; y; \mathcal{C} \vdash_m p * q : \texttt{Mul[QE]} \rightsquigarrow a; b; \gamma}$$

Rule QE-Right handles the symmetric case — a y-variable p times a $\mathbb{Z}_p$ constant q — placing q in the sparse vector b at p's index.

(QE-Bilinear)

$$\frac{\begin{array}{c} X;Y;x;y;\mathcal{C} \vdash p : \texttt{Var}_x\texttt{[ZP]} \\ X;Y;x;y;\mathcal{C} \vdash q : \texttt{Var}_y\texttt{[ZP]} \qquad X;Y;x;y;\mathcal{C} \vdash r : \texttt{Const[ZP]} \\ a = \{\bot\}^{|y|} \qquad b = \{\bot\}^{|x|} \qquad \gamma = \left\{ \begin{array}{ll} \bot & \text{if } (j,k) \neq (\texttt{gi}(p;x), \texttt{gi}(q;y)), \\ r & \text{if } (j,k) = (\texttt{gi}(p;x), \texttt{gi}(q;y)) \end{array} \right\}_{j=1,k=1}^{|x|\times|y|} \end{array}}{X;Y;x;y;\mathcal{C} \vdash_m r * p * q : \texttt{Mul[QE]} \rightsquigarrow a;b;\gamma}$$

Rule QE-Bilinear covers the scaled bilinear term $r * p * q$, where both p and q are variables and r is a $\mathbb{Z}_p$ scalar: the coefficient r is placed at position $(\texttt{gi}(p;x), \texttt{gi}(q;y))$ in the matrix γ.

(QE-Eq)

$$\frac{\begin{array}{c} \forall i, \quad X;Y;x;y;\mathcal{C} \vdash m_i : \texttt{Mul[QE]} \rightsquigarrow a_i;b_i;\gamma_i \\ X;Y;x;y;\mathcal{C} \vdash t : \texttt{Const[ZP]} \qquad a = \curlyvee_i a_i \qquad b = \curlyvee_i b_i \qquad \gamma = \curlyvee_i \gamma_i \end{array}}{X;Y;x;y;\mathcal{C} \vdash_e \sum_i m_i = t : \texttt{Eq[QE]} \rightsquigarrow a;b;\gamma}$$

Rule QE-Eq collects the individual term results using $\curlyvee$, which merges two $\bot$-or-value sparse structures and fails if two non-$\bot$ entries collide at the same index, ensuring every position is determined by at most one term.

Multi Scalar in G1

These rules follow the same three-way split as the QE rules, but one factor now lives in $\mathbb{G}_1$.

(MSE1-Left)

$$\frac{\begin{array}{c} X;Y;x;y;\mathcal{C} \vdash p : \texttt{Const[G1]} \qquad X;Y;x;y;\mathcal{C} \vdash q : \texttt{Var}_y\texttt{[ZP]} \\ a = \left\{ \begin{array}{ll} \bot & \text{if } j \neq \texttt{gi}(q;y), \\ p & \text{if } j = \texttt{gi}(q;y) \end{array} \right\}_{j=1}^{|y|} \qquad b = \{\bot\}^{|X|} \qquad \gamma = \{\bot\}^{|X|\times|y|} \end{array}}{X;Y;x;y;\mathcal{C} \vdash_m p * q : \texttt{Mul[MS1]} \rightsquigarrow a;b;\gamma}$$

Rule MSE1-Left handles a $\mathbb{G}_1$ constant p multiplied by a $\mathbb{Z}_p$ y-variable q, placing p in the sparse vector a.

(MSE1-Right)

$$\frac{\begin{array}{c} X;Y;x;y;\mathcal{C} \vdash p : \texttt{Var[G1]} \qquad X;Y;x;y;\mathcal{C} \vdash q : \texttt{Const[ZP]} \\ a = \{\bot\}^{|y|} \qquad b = \left\{ \begin{array}{ll} \bot & \text{if } j \neq \texttt{gi}(p;X), \\ q & \text{if } j = \texttt{gi}(p;X) \end{array} \right\}_{j=1}^{|X|} \qquad \gamma = \{\bot\}^{|X|\times|y|} \end{array}}{X;Y;x;y;\mathcal{C} \vdash_m p * q : \texttt{Mul[MS1]} \rightsquigarrow a;b;\gamma}$$

Rule MSE1-Right handles a $\mathbb{G}_1$ variable p multiplied by a $\mathbb{Z}_p$ constant q, placing q in b.

$$\frac{\begin{array}{c}\text{(MSE1-Bilinear)}\\ X;Y;x;y;\mathcal{C} \vdash p : \texttt{Var[G1]} \qquad X;Y;x;y;\mathcal{C} \vdash q : \texttt{Var}_y\texttt{[ZP]} \qquad X;Y;x;y;\mathcal{C} \vdash r : \texttt{Const[ZP]}\\ a = \{\bot\}^{|y|} \qquad b = \{\bot\}^{|X|} \qquad \gamma = \left\{ \begin{array}{ll} \bot & \text{if } (j,k) \neq (\texttt{gi}(p;X), \texttt{gi}(q;y)), \\ r & \text{if } (j,k) = (\texttt{gi}(p;X), \texttt{gi}(q;y)) \end{array} \right\}_{j=1,k=1}^{|X|\times|y|}\end{array}}{X;Y;x;y;\mathcal{C} \vdash_m r * p * q : \texttt{Mul[MS1]} \rightsquigarrow a;b;\gamma}$$

Rule MSE1-Bilinear covers the scaled term $r * p * q$ with p a $\mathbb{G}_1$ variable and q a y-variable, recording the scalar r in γ.

$$\frac{\begin{array}{c}\text{(MSE1-Eq)}\\ \forall i, \quad X;Y;x;y;\mathcal{C} \vdash m_i : \texttt{Mul[MS1]} \rightsquigarrow a_i;b_i;\gamma_i\\ X;Y;x;y;\mathcal{C} \vdash t : \texttt{Const[G1]} \qquad a = \bigtriangledown_i a_i \qquad b = \bigtriangledown_i b_i \qquad \gamma = \bigtriangledown_i \gamma_i\end{array}}{X;Y;x;y;\mathcal{C} \vdash_e \sum_i m_i = t : \texttt{Eq[MS1]} \rightsquigarrow a;b;\gamma}$$

Rule MSE1-Eq accumulates all terms via $\bigtriangledown$ and additionally checks that the equation's right-hand side t is a $\mathbb{G}_1$ constant.

Multi Scalar in G2

These rules are symmetric to the $\mathbb{G}_1$ case but with the group roles swapped: $\mathbb{G}_2$ elements now sit on the y-side and are paired with $\texttt{Var}_x\texttt{[ZP]}$ scalars on the x-side.

$$\frac{\begin{array}{c}\text{(MSE2-Left)}\\ X;Y;x;y;\mathcal{C} \vdash p : \texttt{Const[ZP]} \qquad X;Y;x;y;\mathcal{C} \vdash q : \texttt{Var[G2]}\\ a = \left\{ \begin{array}{ll} \bot & \text{if } j \neq \texttt{gi}(q;Y), \\ p & \text{if } j = \texttt{gi}(q;Y) \end{array} \right\}_{j=1}^{|Y|} \qquad b = \{\bot\}^{|x|} \qquad \gamma = \{\bot\}^{|x|\times|Y|}\end{array}}{X;Y;x;y;\mathcal{C} \vdash_m p * q : \texttt{Mul[MS2]} \rightsquigarrow a;b;\gamma}$$

Rule MSE2-Left handles a $\mathbb{Z}_p$ constant p multiplied by a $\mathbb{G}_2$ variable q, placing p in a.

$$\frac{\begin{array}{c}\text{(MSE2-Right)}\\ X;Y;x;y;\mathcal{C} \vdash p : \texttt{Var}_x\texttt{[ZP]} \qquad X;Y;x;y;\mathcal{C} \vdash q : \texttt{Const[G2]}\\ a = \{\bot\}^{|Y|} \qquad b = \left\{ \begin{array}{ll} \bot & \text{if } j \neq \texttt{gi}(p;x), \\ q & \text{if } j = \texttt{gi}(p;x) \end{array} \right\}_{j=1}^{|x|} \qquad \gamma = \{\bot\}^{|x|\times|Y|}\end{array}}{X;Y;x;y;\mathcal{C} \vdash_m p * q : \texttt{Mul[MS2]} \rightsquigarrow a;b;\gamma}$$

Rule MSE2-Right handles a $\texttt{Var}_x\texttt{[ZP]}$ variable p multiplied by a $\mathbb{G}_2$ constant q, placing q in b.

$$\frac{\begin{array}{c}X;Y;x;y;\mathcal{C} \vdash p : \texttt{Var}_x\texttt{[ZP]} \\ X;Y;x;y;\mathcal{C} \vdash q : \texttt{Var[G2]} \qquad X;Y;x;y;\mathcal{C} \vdash r : \texttt{Const[ZP]} \\ a = \{\bot\}^{|Y|} \qquad b = \{\bot\}^{|x|} \qquad \gamma = \left\{ \begin{array}{ll} \bot & \text{if } (j,k) \neq (\texttt{gi}(p;x), \texttt{gi}(q;Y)), \\ r & \text{if } (j,k) = (\texttt{gi}(p;x), \texttt{gi}(q;Y)) \end{array} \right\}_{j=1,k=1}^{|x|\times|Y|}\end{array}}{X;Y;x;y;\mathcal{C} \vdash_m r * p * q : \texttt{Mul[MS2]} \rightsquigarrow a;b;\gamma}\ \text{(MSE2-Bilinear)}$$

Rule MSE2-Bilinear covers the scaled term $r * p * q$ with p a $\texttt{Var}_x\texttt{[ZP]}$ variable and q a $\mathbb{G}_2$ variable, recording r in γ.

$$\frac{\begin{array}{c}\forall i, \quad X;Y;x;y;\mathcal{C} \vdash m_i : \texttt{Mul[MS2]} \rightsquigarrow a_i;b_i;\gamma_i \\ X;Y;x;y;\mathcal{C} \vdash t : \texttt{Const[G2]} \qquad a = \bigtimes_i a_i \qquad b = \bigtimes_i b_i \qquad \gamma = \bigtimes_i \gamma_i\end{array}}{X;Y;x;y;\mathcal{C} \vdash_e \sum_i m_i = t : \texttt{Eq[MS2]} \rightsquigarrow a;b;\gamma}\ \text{(MSE2-Eq)}$$

Rule MSE2-Eq accumulates all terms and requires the right-hand side t to be a $\mathbb{G}_2$ constant.

Pairing Product Equation

These rules handle the full $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ bilinear setting of pairing product equations.

$$\frac{\begin{array}{c}X;Y;x;y;\mathcal{C} \vdash p : \texttt{Const[G1]} \qquad X;Y;x;y;\mathcal{C} \vdash q : \texttt{Var[G2]} \\ a = \left\{ \begin{array}{ll} \bot & \text{if } j \neq \texttt{gi}(q;Y), \\ p & \text{if } j = \texttt{gi}(q;Y) \end{array} \right\}_{j=1}^{|Y|} \qquad b = \{\bot\}^{|X|} \qquad \gamma = \{\bot\}^{|X|\times|Y|}\end{array}}{X;Y;x;y;\mathcal{C} \vdash_m p * q : \texttt{Mul[PPE]} \rightsquigarrow a;b;\gamma}\ \text{(PPE-Left)}$$

Rule PPE-Left pairs a $\mathbb{G}_1$ constant p with a $\mathbb{G}_2$ variable q, placing p in the sparse vector a at q's index.

$$\frac{\begin{array}{c}X;Y;x;y;\mathcal{C} \vdash p : \texttt{Var[G1]} \qquad X;Y;x;y;\mathcal{C} \vdash q : \texttt{Const[G2]} \\ a = \{\bot\}^{|Y|} \qquad b = \left\{ \begin{array}{ll} \bot & \text{if } j \neq \texttt{gi}(p;X), \\ q & \text{if } j = \texttt{gi}(p;X) \end{array} \right\}_{j=1}^{|X|} \qquad \gamma = \{\bot\}^{|X|\times|Y|}\end{array}}{X;Y;x;y;\mathcal{C} \vdash_m p * q : \texttt{Mul[PPE]} \rightsquigarrow a;b;\gamma}\ \text{(PPE-Right)}$$

Rule PPE-Right pairs a $\mathbb{G}_1$ variable p with a $\mathbb{G}_2$ constant q, placing q in b at p's index.

$$\frac{\begin{array}{c}\text{(PPE-Bilinear)}\\ X;Y;x;y;\mathcal{C} \vdash p : \texttt{Var[G1]} \qquad X;Y;x;y;\mathcal{C} \vdash q : \texttt{Var[G2]} \qquad X;Y;x;y;\mathcal{C} \vdash r : \texttt{Const[ZP]}\\ a = \{\bot\}^{|Y|} \qquad b = \{\bot\}^{|X|} \qquad \gamma = \left\{ \begin{array}{ll} \bot & \text{if } (j,k) \neq (\texttt{gi}(p;X), \texttt{gi}(q;Y)), \\ r & \text{if } (j,k) = (\texttt{gi}(p;X), \texttt{gi}(q;Y)) \end{array} \right\}_{j=1,k=1}^{|X|\times|Y|}\end{array}}{X;Y;x;y;\mathcal{C} \vdash_m r * p * q : \texttt{Mul[PPE]} \rightsquigarrow a;b;\gamma}$$

Rule PPE-Bilinear covers the scaled bilinear term $r * p * q$ where both $p \in \mathbb{G}_1$ and $q \in \mathbb{G}_2$ are variables and $r \in \mathbb{Z}_p$ is a scalar, recording r in γ at position $(\texttt{gi}(p;X), \texttt{gi}(q;Y))$.

$$\frac{\begin{array}{c}\text{(PPE-Eq)}\\ \forall i, \quad X;Y;x;y;\mathcal{C} \vdash m_i : \texttt{Mul[PPE]} \rightsquigarrow a_i;b_i;\gamma_i\\ X;Y;x;y;\mathcal{C} \vdash t : \texttt{Const[GT]} \qquad a = \bigtimes_i a_i \qquad b = \bigtimes_i b_i \qquad \gamma = \bigtimes_i \gamma_i\end{array}}{X;Y;x;y;\mathcal{C} \vdash_e \sum_i m_i = t : \texttt{Eq[PPE]} \rightsquigarrow a;b;\gamma}$$

Rule PPE-Eq accumulates all terms via $\curlyvee$ and requires the right-hand side t to be a $\mathbb{G}_T$ constant.

Chapter 5

Groth-Sahai Framework

In this chapter, we introduce the Groth-Sahai framework implementation, which is a general framework for constructing non-interactive zero-knowledge proofs of knowledge (NIZKPoKs) for a wide range of languages. The framework was introduced by Groth and Sahai in [17] and has since been used to construct many efficient NIZKPoKs for various languages. The framework is based on bilinear groups and the notion of witness indistinguishability, which we define together with the Groth-Sahai proof system in Section 2.5.

5.1 Eye-ball correspondence

In this section, we are going to compare the code snippet from the `framework.py` file with the corresponding algorithm in the paper document.

5.1.1 Setup Algorithm

```
from elements import (
    Element,
    ZpElement,
    G1Element,
    G2Element,
    GTElement,
    # pairing function
    # as method on G1Element
    g1,
    g2,
)
```

Setup:

$\text{gk} := (\mathbf{p}, G_1, G_2, G_T, e, \mathcal{P}_1, \mathcal{P}_2) \leftarrow \mathcal{G}_{\text{SXDH}}(1^k)$.

Figure 5.1: Setup algorithm comparation.

5.1.2 Soundness String Generation Algorithm

```
def new_sound(trapdoor=None):
    if not trapdoor:
        trapdoor = {'alpha1': ZpElement.random(),
                    'alpha2': ZpElement.random(),
                    't1': ZpElement.random(),
                    't2': ZpElement.random()}

    alpha1 = trapdoor['alpha1']
    alpha2 = trapdoor['alpha2']
    t1 = trapdoor['t1']
    t2 = trapdoor['t2']
    u_1 = np.array([g1, alpha1 * g1])
    v_1 = np.array([g2, alpha2 * g2])
    u_2 = t1 * u_1
    v_2 = t2 * v_1

    return CRS(u_1, u_2, v_1, v_2, trapdoor=trapdoor)
```

Soundness string: On input gk, return $\sigma := (u_1, u_2, v_1, v_2)$ where $u_2 = t_1 u_1$ and $v_2 = t_2 v_1$ for random $t_1, t_2 \leftarrow \mathbb{Z}_p$.

Figure 5.2: Soundness string algorithm comparation.

5.1.3 Witness-indistinguishability String Generation Algorithm

```
def new_wi(trapdoor=None):
    if  not trapdoor:
        trapdoor = {'alpha1': ZpElement.random(),
                    'alpha2': ZpElement.random(),
                    't1': ZpElement.random(),
                    't2': ZpElement.random()}

    alpha1 = trapdoor['alpha1']
    alpha2 = trapdoor['alpha2']
    t1 = trapdoor['t1']
    t2 = trapdoor['t2']
    u_1 = np.array([g1, alpha1 * g1])
    v_1 = np.array([g2, alpha2 * g2])
    u_2 = t1 * u_1 - np.array([G1Element.zero(), g1])
    v_2 = t2 * v_1 - np.array([G2Element.zero(), g2])

    return CRS(u_1, u_2, v_1, v_2, trapdoor=trapdoor)
```

Witness-indistinguishability string: On input gk, return $\sigma := (u_1, u_2, v_1, v_2)$ where $u_2 = t_1 u_1 - (\mathcal{O}, \mathcal{P}_1)$ and $v_2 = t_2 v_1 - (\mathcal{O}, \mathcal{P}_2)$ for random $t_1, t_2 \leftarrow \mathbb{Z}_p$.

Figure 5.3: Witness-indistinguishability algorithm comparation.

5.1.4 Proof Algorithm

```
def proof(crs: CRS, eqs: equations, X, Y, x, y):
    m = len(X)
    m_prime = len(x)
    n = len(Y)
    n_prime = len(y)

    R = np.array([[ZpElement.random() for   in range(2)] for   in range(m)]) # shape (m, 2) of Zp
    r = np.array([ZpElement.random() for _ in range(m_prime)]) # shape (m,) of Zp

    c = X.iota_1(crs) + R@crs.u if len(X) > 0 else NamedArray([])
    c_prime = x.iota_prime_1(crs) + r@crs.u1 if len(x) > 0 else NamedArray([])
    S = np.array([[ZpElement.random() for   in range(2)] for   in range(n)]) # shape (m, 2) of Zp
    s = np.array([ZpElement.random() for   in range(n_prime)]) # shape (m,) of Zp
    d = Y.iota_2(crs) + S@crs.v if len(Y) > 0 else NamedArray([])
    d_prime = y.iota_prime_2(crs) + s@crs.v1 if len(y) > 0 else NamedArray([])
    ...
```

Figure 5.4: Proof algorithm code (part 1).

On input gk, σ, a set of equations and a witness $\vec{\mathcal{X}}, \vec{\mathcal{Y}}, \vec{x}, \vec{y}$, do:

1. Commit to the group elements $\vec{\mathcal{X}} \in G_1^m$ and the scalars $\vec{x} \in \mathbb{Z}_p^{m'}$ as

   $$\vec{c} := \iota_1(\vec{x}) + R\vec{u}, \quad c' := \iota_1(x) + r^\top \vec{u}_1$$

   where $R \leftarrow \mathrm{Mat}_{m\times 2}(\mathbb{Z}_p)$, $\vec{r} \leftarrow \mathbb{Z}_p^{m'}$.

   Commit to the group elements $\vec{\mathcal{Y}} \in G_2^n$ and the scalars $\vec{y} \in \mathbb{Z}_p^{n'}$ as

   $$\vec{d} := \iota_2(\vec{y}) + S\vec{v}, \quad d' := \iota_2(y) + s^\top \vec{v}_1$$

   where $S \leftarrow \mathrm{Mat}_{n\times 2}(\mathbb{Z}_p)$, $\vec{s} \leftarrow \mathbb{Z}_p^{n'}$.

Figure 5.5: Proof algorithm description (part 1).

```
def proof(crs: CRS, eqs: equations, X, Y, x, y):
    ...

    pis = []
    thetas = []
    for eq in eqs.ppe:
        A = eq.a
        B = eq.b
        Gamma = np.array(eq.Gamma)
        T = random_zp(2,2)
        if len(X) == 0:
            pi = np.array([[G2Element.zero(), G2Element.zero()], [G2Element.zero(), G2Element.zero()]])
            theta = S.T @ np.array(list(map(crs.iota_1, A)))
        elif len(Y) == 0:
            pi = R.T @ np.array(list(map(crs.iota_2, B)))
            theta = np.array([[G1Element.zero(), G1Element.zero()], [G1Element.zero(), G1Element.zero()]])
        else:
            pi = R.T @ np.array(list(map(crs.iota_2, B))) + R.T @ Gamma @ Y.iota_2(crs) + (R.T @ Gamma @ S - T.T) @ crs.v
            theta = S.T @ np.array(list(map(crs.iota_1, A))) + S.T @ Gamma.T @ X.iota_1(crs) + T @ crs.u

        pis.append(UnamedArray(pi))
        thetas.append(UnamedArray(theta))

    ...
```

Figure 5.6: Proof algorithm code (part 2).

2. For each pairing product equation $(\vec{A} \cdot \vec{\mathcal{Y}})(\vec{\mathcal{X}} \cdot \vec{B})(\vec{\mathcal{X}} \cdot \Gamma \vec{\mathcal{Y}}) = t_T$, the proof is for random $T \leftarrow \mathrm{Mat}_{2\times 2}(\mathbb{Z}_p)$:

$$\vec{\pi} := R^\top \iota_2(\vec{B}) + R^\top \Gamma \iota_2(\vec{\mathcal{Y}}) + (R^\top T S - T^\top)\vec{v}$$

$$\vec{\theta} := S^\top \iota_1(\vec{A}) + S^\top T^\top \iota_1(\vec{\mathcal{X}}) + T\vec{u}.$$

For each linear equation $\vec{A} \cdot \vec{\mathcal{Y}} = t_T$ we use $\vec{\pi} := \vec{0}$ and $\vec{\theta} := S^\top \iota_1(\vec{A})$. There is a bijective correspondence between $S^\top \vec{A} = p_1(\vec{\theta})$ and $\vec{\theta} = \iota_1(S^\top \vec{A})$. The proof can therefore be communicated by sending $S^\top \vec{A}$, which consists of two group elements in G_1.

For each linear equation $\vec{\mathcal{X}} \cdot \vec{B} = t_T$ we use $\vec{\pi} := R^\top \iota_2(\vec{B})$ and $\vec{\theta} := \vec{0}$. As above, the proof can be communicated by sending the two group elements $R^\top \vec{B}$ in G_2.

Figure 5.7: Proof algorithm description (part 2).

```
def proof(crs: CRS, eqs: equations, X, Y, x, y):
    ...

    for eq in eqs.ms1:
        A = eq.A
        b = eq.b
        Gamma = eq.Gamma
        T = random_zp(1,2)
        if len(X) == 0:
            pi = np.array([[G2Element.zero(), G2Element.zero()]])
            theta = S.T @ np.array(list(map(crs.iota_1, A)))
        elif len(y) == 0:
            pi = R.T @ np.array(list(map(crs.iota_2, B)))
            theta = np.array([G1Element.zero(), G1Element.zero()])
        else:
            pi = R.T @ crs.iota_prime_2(b) + R.T @ Gamma @ crs.iota_prime_2(y) + (R.T @ Gamma @ s - T.T) @ crs.v1
            theta = s.T @ crs.iota_1(A) + s.T @ Gamma.T @ crs.iota_1(X) + T @ crs.u
        pis.append(UnamedArray(pi))
        thetas.append(UnamedArray(theta))

    ...
```

Figure 5.8: Proof algorithm code (part 3).

3. For each multi-scalar multiplication equation $(\vec{A} \cdot \vec{\mathcal{Y}} + \vec{x} \cdot \vec{B} + \vec{x}^\top \Gamma \vec{\mathcal{Y}} = T_1)$ in G_1, the proof is for random $T \leftarrow \mathrm{Mat}_{1\times 2}(\mathbb{Z}_p)$:

$$\vec{\pi} := R^\top \iota_2(\vec{B}) + R^\top \Gamma \iota_2(\vec{\mathcal{Y}}) + (R^\top T S - T^\top)\vec{v}$$

$$\vec{\theta} := S^\top \iota_1(\vec{A}) + S^\top T^\top \iota_1(\vec{\mathcal{X}}) + T\vec{u}.$$

For each linear equation $\vec{a} \cdot \vec{\mathcal{Y}} = T_2$ the proof is $\vec{\pi} := 0$ and $\vec{\theta} := S^\top \iota_1(\vec{a})$. There is a bijective correspondence between $S^\top \vec{a} = p_1(\vec{\theta})$ and $\vec{\theta} = \iota_1(S^\top \vec{a})$. The proof can therefore be communicated by sending $S^\top \vec{a}$, which consists of two field elements.

For each linear equation $\vec{x} \cdot \vec{B} = T_2$ the proof is $\vec{\pi} := R^\top \iota_2(\vec{B})$ and $\vec{\theta} := 0$. As above, the proof can be communicated by sending the single group element $R^\top \vec{B}$.

Figure 5.9: Proof algorithm (part 3).

```
def proof(crs: CRS, eqs: equations, X, Y, x, y):
    ...

    for eq in eqs.ms2:
        a = eq.a
        B = eq.B
        Gamma = eq.Gamma
        T = random_zp(2,1)
        if len(x) == 0:
            pi = np.array([G2Element.zero(), G2Element.zero()])
            theta = S.T @ np.array(list(map(crs.iota_1, a)))
        elif len(Y) == 0:
            pi = r.T @ np.array(list(map(crs.iota 2, B)))
            theta = np.array([[G1Element.zero(), G1Element.zero()]])
        pi = r.T * crs.iota 2(B) + r.T * Gamma * crs.iota 2(Y) + (r.T * Gamma * S - T.T) * crs.v
        theta = S.T * crs.iota_prime_1(a) + S.T * Gamma.T * crs.iota_prime_1(x) + T * crs.u1
        pis.append(UnamedArray(pi))
        thetas.append(UnamedArray(theta))

    ...
```

Figure 5.10: Proof algorithm code (part 4).

4. For each multi-scalar multiplication equation $\vec{a} \cdot \vec{\mathcal{Y}} + \vec{x} \cdot \vec{B} + \vec{x}^{\top} \Gamma \vec{\mathcal{Y}} = T_2$ in G_2, the proof is for random $T \leftarrow \mathrm{Mat}_{2\times 1}(\mathbb{Z}_p)$:

$$\vec{\pi} := \vec{r}^{\top}\iota_2(\vec{B}) + \vec{r}^{\top}\Gamma\iota_2(\vec{\mathcal{Y}}) + (\vec{r}^{\top}\Gamma S - T^{\top})\vec{v}$$

$$\vec{\theta} := S^{\top}\iota_1(\vec{a}) + S^{\top}T^{\top}\iota_1(\vec{x}) + T\vec{u}.$$

For each linear equation $\vec{a} \cdot \vec{y} = T_2$ the proof is $\vec{\pi} := 0$ and $\vec{\theta} := S^{\top}\iota_1(\vec{a})$. The proof can therefore be communicated by sending $S^{\top}\vec{a}$.

For each linear equation $\vec{x} \cdot \vec{B} = T_2$ the proof is $\vec{\pi} := \vec{r}^{\top}\iota_2(\vec{B})$ and $\vec{\theta} := 0$. As above, the proof can be communicated by sending the single group element $\vec{r}^{\top}\vec{B}$.

Figure 5.11: Proof algorithm (part 4).

```
def proof(crs: CRS, eqs: equations, X, Y, x, y):
    ...

    for eq in eqs.qe:
        a = eq.a
        b = eq.b
        Gamma = np.array(eq.Gamma)
        T = ZpElement.random()
        if len(x) == 0:
            pi = np.array([G2Element.zero(), G2Element.zero()])
            theta = s.T @ np.array(list(map(crs.iota_prime_1, a)))
        elif len(y) == 0:
            pi = r.T @ np.array(list(map(crs.iota_prime_2, b)))
            theta = np.array([G1Element.zero(), G1Element.zero()])
        else:
            pi = r.T @ crs.iota_prime_2(b) + r.T @ Gamma @ crs.iota_prime_2(y) + (r.T @ Gamma @ s - T) @ crs.v1
            theta = s.T @ crs.iota_prime_1(a) + s.T @ Gamma.T @ crs.iota_prime_1(x) + T @ crs.u1

        pis.append(UnamedArray(pi))
        thetas.append(UnamedArray(theta))

    return Proof(c, c_prime, d, d_prime, pis, thetas, None)
```

Figure 5.12: Proof algorithm code (part 5).

5. For each quadratic equation $\vec{a} \cdot \vec{y} + \vec{x} \cdot \vec{b} + \vec{x}^{\top} \Gamma \vec{y} = t$ in $\mathbb{Z}_p$, the proof is for random $T \leftarrow \mathbb{Z}_p$:

$$\pi := \vec{r}^{\top} \iota_2(\vec{b}) + \vec{r}^{\top} \Gamma \iota_2(\vec{y}) + (\vec{r}^{\top} \Gamma S - T)\vec{v}$$

$$\theta := \vec{s}^{\top} \iota_1(\vec{a}) + \vec{s}^{\top} T^{\top} \iota_1(\vec{x}) + T\vec{u}.$$

   For each linear equation $\vec{a} \cdot \vec{y} = t$ we use $\pi := 0$ and $\theta := \vec{s}^{\top} \iota_1(\vec{a})$. The proof can therefore be communicated by sending $\vec{s}^{\top} \vec{a}$.

   For each linear equation $\vec{x} \cdot \vec{b} = t$ we use $\pi := \vec{r}^{\top} \iota_2(\vec{b})$. As above, the proof can be communicated by sending the single field element $\vec{r}^{\top} \vec{b}$.

Figure 5.13: Proof algorithm (part 5).

5.1.5 Verify Algorithm

```
def verify(crs: CRS, eqs: equations, c, c_prime, d, d_prime, pis, thetas):
    for eq, pi, theta in zip(eqs, pis, thetas):
        a = UnamedArray(eq.a)
        b = UnamedArray(eq.b)
        Gamma = UnamedArray(eq.Gamma)
        t = eq.t

        if eq.etype == 3:
            lhs = (a.iota_1(crs)).b_pair(d) * c.b_pair(b.iota_2(crs)) * c.b_pair(Gamma @ d)
            rhs = crs.iota_t(t) * crs.u.b_pair(pi) * theta.b_pair(crs.v)
            if not np.all(lhs == rhs): return False

        elif eq.etype == 1:
            lhs = (a.iota_1(crs)).b_pair(d_prime) * c.b_pair(b.iota_prime_2(crs)) * c.b_pair(Gamma @ d_prime)
            rhs = crs.iota_t_tilde(t) * crs.u.b_pair(pi) * theta.b_pair(crs.v[0])
            if not np.all(lhs == rhs): return False
        elif eq.etype == 2:
            lhs = (a.iota_prime_1(crs)).b_pair(d) * c_prime.b_pair(b.iota_2(crs)) * c_prime.b_pair(Gamma @ d)
            rhs = crs.iota_t_hat(t) * crs.u[0].b_pair(pi) * theta.b_pair(crs.v)
            if not np.all(lhs == rhs): return False
        elif eq.etype == 0:
            lhs = (a.iota_prime_1(crs)).b_pair(d_prime) * c_prime.b_pair(b.iota_prime_2(crs)) * c_prime.b_pair(Gamma @ d_prime)
            rhs = crs.iota_t_prime(t) * crs.u[0].b_pair(pi) * theta.b_pair(crs.v[0])
            if not np.all(lhs == rhs): return False
    return True
```

Figure 5.14: Verify algorithm code.

On input (gk, σ), a set of equations and a proof $\vec{c}, \vec{d}, c', d', \{\vec{\pi}_i, \vec{\theta}_i\}_{i=1}^N$, do:

1. For each pairing product equation

$$(\vec{A} \cdot \vec{Y})(\vec{X} \cdot \vec{B})(\vec{X} \cdot \Gamma\vec{Y}) = t_T$$

with proof $(\vec{\pi}, \vec{\theta})$, check that

$$\iota_1(\vec{A}) \cdot \vec{d} + c' \cdot \iota_2(\vec{B}) + c' \cdot \Gamma\vec{d} = \iota_T(t_T) + \vec{u} \cdot \vec{\pi} + \vec{\theta} \cdot \vec{v}.$$

2. For each multi-scalar equation

$$\vec{A} \cdot \vec{Y} + \vec{x} \cdot \vec{B} + \vec{x}^\top \Gamma \vec{Y} = T_1$$

in G_1 with proof $(\vec{\pi}, \vec{\theta})$, check that

$$\iota_1(\vec{A}) \cdot \vec{d} + c' \cdot \iota_2(\vec{B}) + c' \cdot \Gamma\vec{d} = \iota_T(T_1) + \vec{u} \cdot \vec{\pi} + F(\vec{\theta}, v_1).$$

3. For each multi-scalar equation

$$\vec{a} \cdot \vec{Y} + \vec{x} \cdot \vec{B} + \vec{x}^\top \Gamma \vec{Y} = T_2$$

in G_2 with proof $(\vec{\pi}, \vec{\theta})$, check that

$$\iota_1(\vec{a}) \cdot \vec{d} + c' \cdot \iota_2(\vec{B}) + c' \cdot \Gamma\vec{d} = \iota_T(T_2) + F(\vec{u}, \vec{\pi}) + \vec{\theta} \cdot \vec{v}.$$

4. For each quadratic equation

$$\vec{a} \cdot \vec{y} + \vec{x} \cdot \vec{b} + \vec{x}^\top \Gamma \vec{y} = t$$

in $\mathbb{Z}_p$ with proof $(\vec{\pi}, \vec{\theta})$, check that

$$\iota_1(\vec{a}) \cdot \vec{d} + c' + \iota_2(\vec{b}) \cdot \vec{d'} = \iota_T(t) + F(\vec{u}_1, \vec{\pi}) + F(\vec{\theta}, v_1).$$

Figure 5.15: Verify algorithm description.

5.2 Python Decorator

In this section, we introduce a Python decorator that enables automatic extraction of Groth-Sahai equations from Python functions. This decorator analyzes the function's bytecode at runtime to identify variables and constructs the appropriate Groth-Sahai equations based on the function's operations.

The decorator looks for a special variable named `GS_STRING` which provides the framework configuration. It then performs dynamic analysis of the function to extract variable names of elements in the proof system.

For example, a function from BLS signature verification is decorated with `@gsfy` could look like:

```
@gsfy
def verify(self, vk: G2Element, m: str, signature: G1Element) -> bool:
    g2 = self.g2
    lhs = signature.pair(g2)
    rhs = G1Element.hash_from_string(m).pair(vk)
    GS_STRING = """
    variables:
        signature: G1
    constants:
        g2: G2
        rhs: GT
    equations:
        signature * g2 = rhs
    """
    return lhs == rhs
```

Figure 5.16: BLS signature verification function decorated with `@gsfy`.

The decorator will analyze this function and extract variables and constants defined in the function. It then constructs the appropriate Groth-Sahai proof system automatically. This provides a convenient high-level interface for specifying equations while handling the low-level details of the framework internally.

The decorator supports all equation types implemented in the framework. This allows developers to write proof systems in a natural mathematical notation while the decorator handles translation to the underlying Groth-Sahai framework.

5.3 Running the Framework

The framework is implemented in Python and can be run using the `main` script. The script provides three main commands: `prove`, `verify`, and `crs`. Each command supports various arguments to customize its behavior.

5.3.1 Command-Line Interface

Environment Variables

The framework supports the following environment variables for default configurations:

- `GS_INPUT`: Default input file path (default: "prove.gs")
- `GS_DEFINITIONS`: Default definitions file path (default: "prove.json")
- `GS_OUTPUT`: Default output file path (default: "prove.py")
- `GS_PROOF_OUTPUT`: Default proof output file path (default: "verify.json")
- `GS_CRS`: Default CRS file path (default: "crs.json")
- `GS_CRS_TYPE`: Default CRS type (default: "sound")

Prove Command

The prove command generates proofs for the Groth-Sahai framework:

```
python -m gskit prove [-h] [-i INPUT] [-d DEFINITIONS] [-c CRS]
                      [-o OUTPUT] [-p PROOF_OUTPUT] [-f]
```

Arguments:

- `-i, --input`: Input file in .gs format
- `-d, --definitions`: Definitions file in JSON format
- `-c, --crs`: Common Reference String (CRS) file
- `-o, --output`: Output Python file
- `-p, --proof-output`: Proof output definitions file
- `-f, --forward`: Enable forward mode to execute the proof

Verify Command

The verify command validates proofs:

```
python -m gskit verify [-h] [-i INPUT] [-d DEFINITIONS]
                       [-c CRS] [-o OUTPUT] [-f]
```

Arguments:

- `-i, --input`: Input file in .gs format
- `-d, --definitions`: Definitions file in JSON format
- `-c, --crs`: Common Reference String (CRS) file
- `-o, --output`: Output Python file
- `-f, --forward`: Enable forward mode to execute the verification

CRS Generation Command

The crs command generates a new Common Reference String:

```
python -m gskit crs [-h] [-c CRS] [-t TYPE]
```

Arguments:

- `-c, --crs`: Output file for the CRS
- `-t, --type`: CRS type ("sound" or "wi")

5.3.2 Compilation Process

The framework uses a compilation process to generate proofs and verifications as shown in Figures 5.17 and 5.18. These graphs illustrate the transformation of the input files through various stages of the framework.

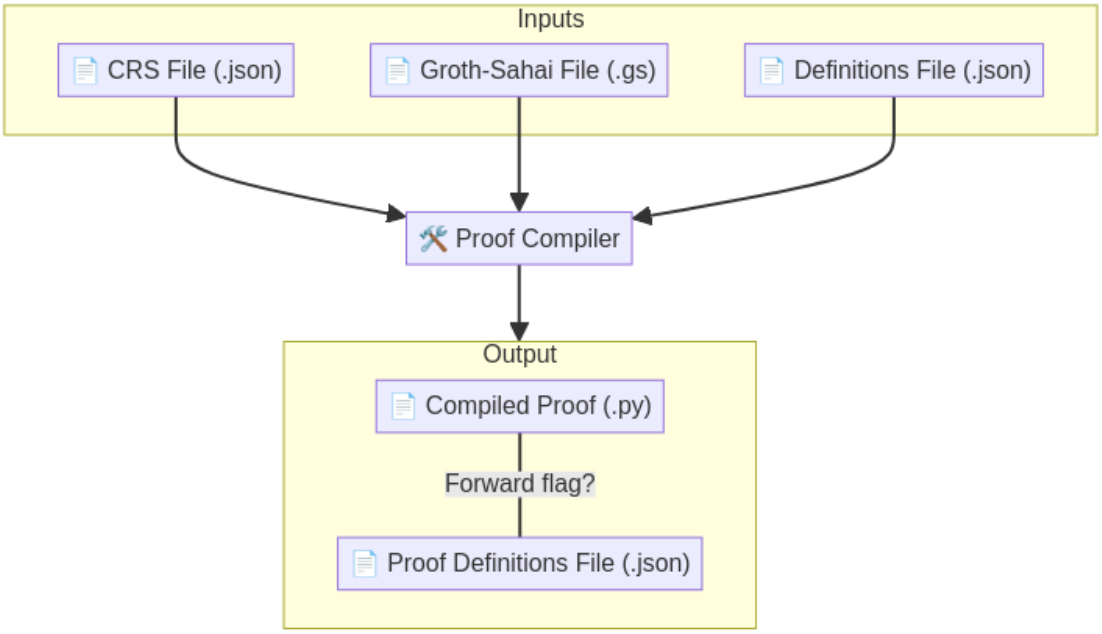

Figure 5.17: Proof compiler graph.

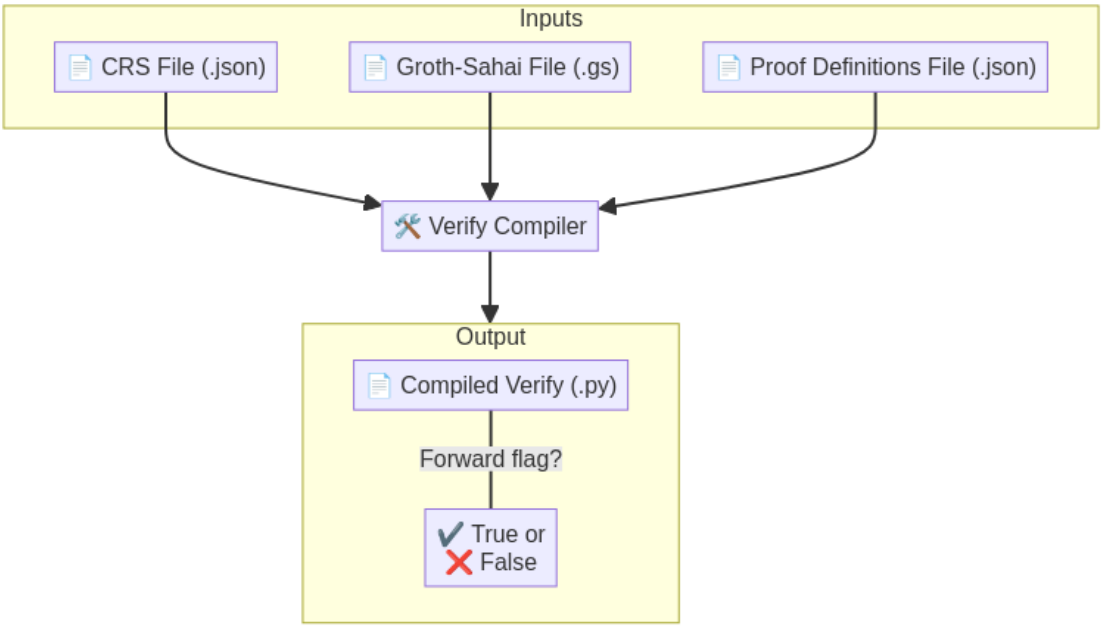

Figure 5.18: Verify compiler graph.

5.4 Evaluation

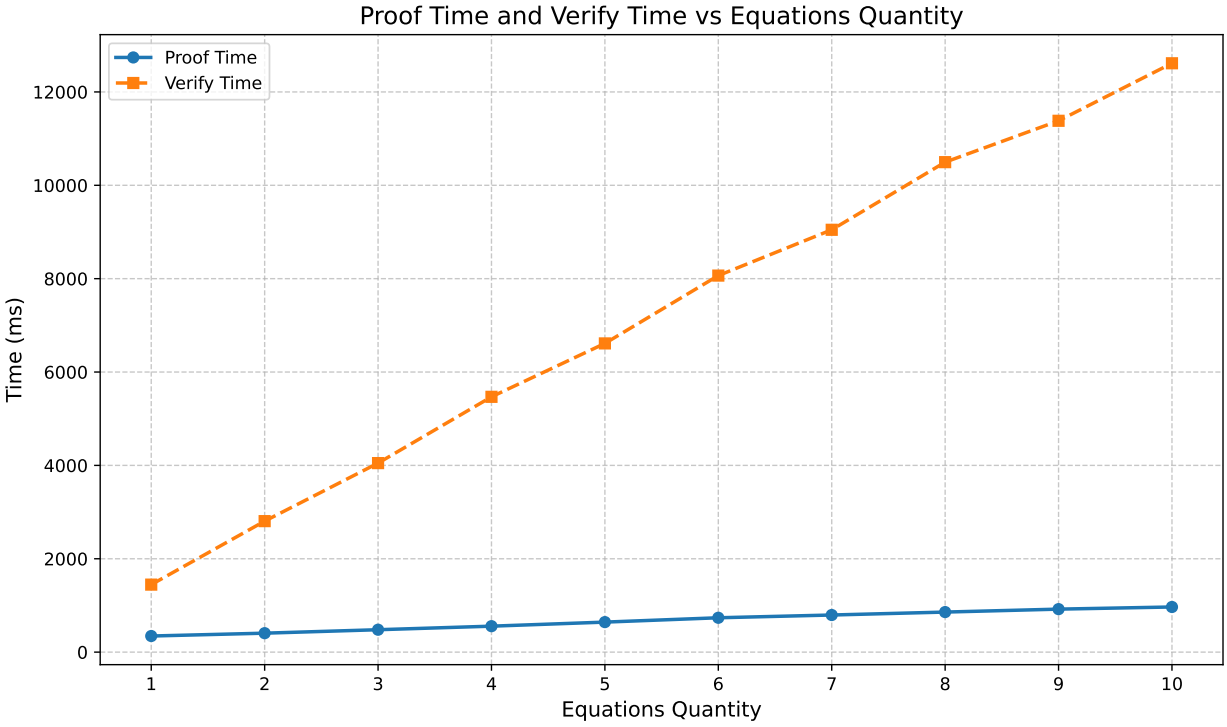

Figure 5.19: Performance varying number of equations

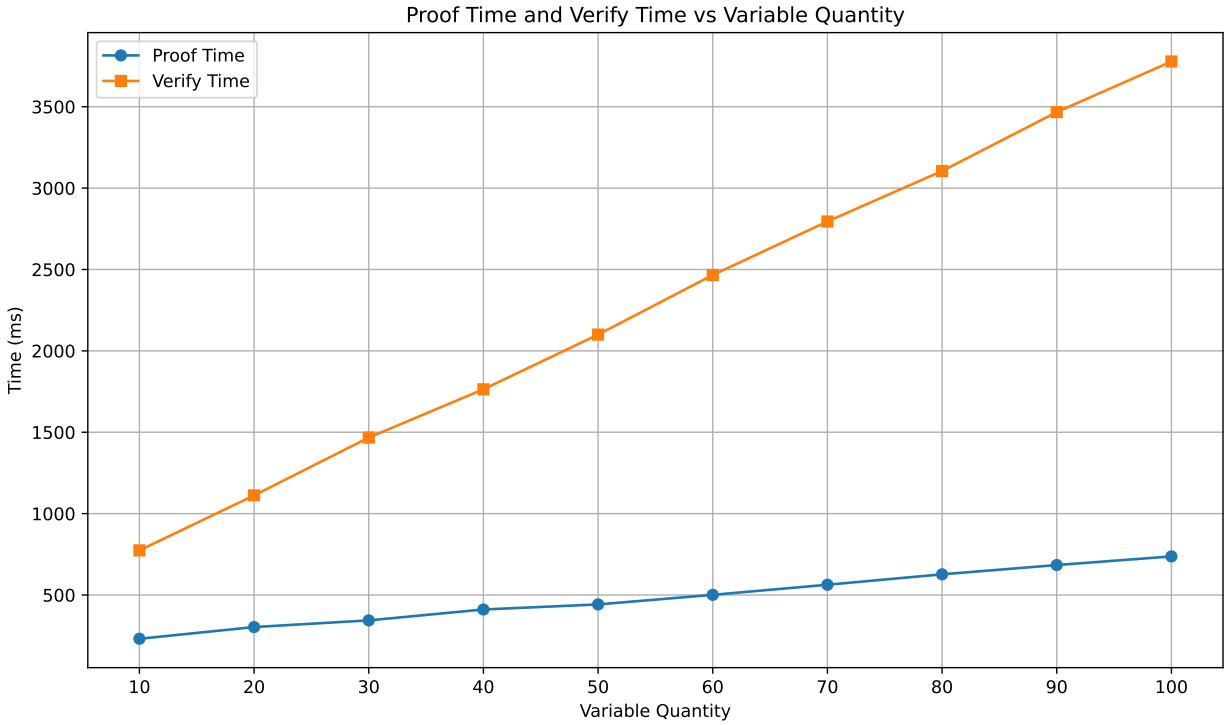

Figure 5.20: Performance varying number of variables

To evaluate the performance of our Groth-Sahai framework implementation, we conducted experiments measuring the execution time under varying conditions.

As shown in Figure 5.19, we analyzed how the framework's performance scales with an increasing number of equations while keeping the number of variables constant. This helps us understand the computational overhead introduced by additional equations in the proof system.

Additionally, Figure 5.20 demonstrates the impact of increasing the number of variables while maintaining a fixed number of equations. This measurement is crucial as it reflects how the system handles growing complexity in terms of witness size.

Both graphs exhibit a linear growth pattern, indicating that our implementation scales efficiently with both the number of equations and variables. This linear scaling is particularly important for practical applications, as it ensures predictable performance characteristics when dealing with larger proof systems.

Chapter 6

Study Cases

In this chapter, we present two case studies to evaluate our Groth-Sahai framework implementation: the Boneh-Lynn-Shacham (BLS) signature scheme and the Attribute-Based Signature with User-Controlled Linkability (ABS-UCL).

The BLS signature scheme is a simple yet powerful digital signature scheme that leverages bilinear pairings. It provides short signatures and strong security guarantees. We use it as our first case study to demonstrate how our framework can handle basic pairing product equations and prove knowledge of valid signatures.

The ABS-UCL scheme represents a more complex application, combining attribute-based signatures with user-controlled linkability. This advanced cryptographic primitive allows users to sign messages based on their attributes while maintaining control over signature linkability. We use it as our second case study to showcase how our framework can handle more sophisticated proof statements involving multiple variables and equations.

Through these case studies, we aim to:

- Demonstrate the framework's ability to handle both simple and complex zero-knowledge proofs
- Evaluate the performance characteristics under different types of equations
- Show how the Python decorator simplifies the creation of zero-knowledge proofs
- Provide concrete examples of real-world applications

6.1 BLS

The Boneh-Lynn-Shacham (BLS) signature scheme [11] is a digital signature scheme that leverages bilinear pairings to create short signatures while maintaining strong security properties. The scheme consists of three main algorithms:

- **Key Generation**: Generates a secret key $sk \leftarrow \mathbb{Z}_p$ and a verification key $vk = sk \cdot g_2$ where g_2 is a generator of $\mathbb{G}_2$
- **Signing**: Given a message m, computes $\sigma = sk \cdot \mathcal{H}(m)$ where $\mathcal{H}$ is a hash function mapping messages to $\mathbb{G}_1$ elements
- **Verification**: Checks if $e(\sigma, g_2) = e(\mathcal{H}(m), vk)$ using the bilinear pairing e

As shown in Figure 5.16, our framework can automatically extract the verification equation and construct a zero-knowledge proof of signature knowledge. The proof allows a prover to convince a verifier that they know a valid signature σ satisfying the verification equation, without revealing the actual signature.

The BLS example demonstrates how our framework handles pairing product equations naturally. The `@gsfy` decorator extracts the verification equation from the Python code and automatically constructs the appropriate Groth-Sahai proof system. This provides a convenient high-level interface while handling the low-level cryptographic details internally.

6.2 ABS-UCL

In this section, we present the integration of ABS-UCL into our framework. ABS-UCL is the most demanding of our case studies: a single signature requires proving, in zero knowledge, a conjunction of statements drawn from several independent cryptographic sub-protocols. We use it to evaluate whether the modular composition supported by GSkit scales to a realistic, multi-equation scheme, and we compare the resulting implementation against an earlier version of the same scheme built directly on a low-level Groth-Sahai library.

6.2.1 Scheme Overview

ABS-UCL is a hybrid scheme that combines an attribute-based signature with user-controlled linkability. It operates in two phases. In the attribute-authority phase, each authority (AA) issues attribute secret keys to a user by signing the user's public key together with an attribute, using a re-randomizable structure-preserving signature (RPSPS). In the signing phase, the user produces a signature over a message that proves, without revealing identity or the specific attributes used, that: (i) the attributes they hold satisfy the access policy, expressed as a monotone span program (MSP); (ii) a linking token (LIT) is correctly formed under the user's secret key and the chosen recipient, which is what enables controlled linkability; and (iii) a pseudo-attribute (PSDO), signed under an FBB signature, is present so that the policy can always be satisfied through a distinguished branch. The verifier checks the resulting Groth-Sahai proof, and the linking and identification procedures operate on the linking token.

The scheme is therefore naturally decomposed into four sub-protocols—RPSPS, LIT, FBB (PSDO), and MSP—each contributing its own set of Groth-Sahai equations to the overall statement. This structure is precisely what motivates a modular treatment.

6.2.2 Modular Construction

The defining feature of our integration is that each sub-protocol declares its own Groth-Sahai verification logic *once*, as a declarative specification attached to its verification method through the `@gsfy` decorator, and the top-level scheme merely *composes* these specifications over a shared proof context (`GSContext`). When the user signs, `ABSUCL.sign()` walks the attributes of the policy and, for each one, adds the corresponding sub-protocol equations to the context through a single call, supplying the public values and witnesses; the framework then compiles a single proof covering the whole conjunction. Both the proving code and the verification code are generated from the same specification, so the prover's and the verifier's view of every equation are guaranteed to coincide.

This contrasts sharply with the manual approach taken in the earlier implementation. There, every Groth-Sahai equation is assembled by hand inside the signing routine, using explicit equation objects (`QEquation`, `MS1Equation`, `PPEquation`) and pairing-map objects (`AMapLeft`, `AMapRight`, `AMapBoth`), and then the same block of equations is assembled *again*, almost identically, inside the verification routine. Figures 6.1 and 6.2 show the same RPSPS attribute verification expressed both ways.

```
# --- inside sign(), repeated for each attribute a ---
S_amap1 = AMapBoth(variables[f"S_{a}"], variables[f"z_{a}"])
S_amap2 = AMapLeft(variables[f"Ssmile_{a}"], Constant(ZpElement.init(-1)))
S_eq    = MS1Equation(f"S_{a}", [S_amap1, S_amap2], Constant(G1Element.zero()))

R_amap1 = AMapRight(Constant(sk_ida[a][0]), variables[f"z_{a}"])
R_amap2 = AMapLeft(variables[f"Rsmile_{a}"], Constant(ZpElement.init(-1)))
R_eq    = MS1Equation(f"R_{a}", [R_amap1, R_amap2], Constant(G1Element.zero()))

V_amap1 = AMapBoth(variables[f"Rsmile_{a}"], variables["Ftilda"])
V_amap2 = AMapLeft(variables[f"Rsmile_{a}"], Constant(self.vk_attrs[a][0]))
V_amap3 = AMapLeft(variables[f"Rsmile_{a}"], Constant(attr * self.vk_attrs[a][1]))
V_amap4 = AMapLeft(variables[f"Ssmile_{a}"], Constant(~self.vk_attrs[a][2]))
V_eq    = PPEquation(f"V_{a}", [V_amap1, V_amap2, V_amap3, V_amap4],
                     Constant(GTElement.zero()))
# ... this whole block is duplicated verbatim inside verify()
```

Figure 6.1: Manual construction (earlier implementation): the RPSPS attribute verification built by hand with explicit equation and pairing-map objects. The identical block is written a second time in the verification routine.

```
@gsfy(witness=['z', 'S', 'S_tilde', 'R_tilde'])
def z_is_zero_or_verify_absucl(self, vk, Ftilda, R, signature_S, z, attr):
    GS_STRING = """
    variables:
        z: ZP
        S: G1
        S_tilde: G1
        R_tilde: G1
    constants:
        R: G1
        z_minus_one: ZP
        Ftilda: G2
        X: G2
        attr_Y: G2
        Z_neg: G2
    equations:
        z * S + z_minus_one * S_tilde = g1_zero
        z * R + z_minus_one * R_tilde = g1_zero
        S_tilde * Z_neg + R_tilde * Ftilda + R_tilde * X
            + R_tilde * attr_Y = gt_zero
    """
```

Figure 6.2: Modular construction (GSkit): the same three equations declared once inside the RPSPS sub-protocol. The `@gsfy` decorator compiles both proving and verification code from this single specification.

In the modular version, `ABSUCL.sign()` no longer constructs these equations at all; it composes the sub-protocol specification into the shared context with a single call per attribute, as shown below, and the verification path is generated automatically from the same source.

```
self.psps.z_is_zero_or_verify_absucl.gs_add(
    ctx, self.psps, vk=self.vk_attrs[a], Ftilda=Ftilda,
    R=R, signature_S=S, z=z, attr=attr_zp, aliases={...})
```

6.2.3 Code Comparison

We quantify the difference between the two implementations along two axes. Table 6.1 reports the total developer-written lines of code (excluding blank lines and comments) for the scheme and its sub-protocols. Notably, the modular implementation is *not* smaller by this raw measure: in the manual version the Groth-Sahai equations live almost entirely inside `absucl`, while the sub-protocols are plain cryptographic code; in the modular version each sub-protocol additionally carries its own Groth-Sahai specification, which redistributes—and slightly increases—the total line count.

Module (developer-written)	Manual	Modular
`absucl` (orchestration)	297	236
`rpsps`	43	95
`lit`	21	67
`msp`	34	75
`ds` (FBB)	25	67
Total (lines of code)	**420**	**540**

Table 6.1: Total developer-written lines of code (blank lines and comments excluded).

The raw line count, however, is the wrong measure of the cost the framework is designed to reduce. The relevant quantity is the amount of *Groth-Sahai equation-construction code* the developer must write and maintain, and how many times each equation is expressed. Table 6.2 isolates this. In the manual implementation, the equation construction occupies 85 lines in the signing routine and a near-identical 84 lines in the verification routine—169 lines in total, roughly half of which are pure duplication—and involves the hand-construction of 60 low-level equation and pairing-map objects. In the modular implementation, the same statement is expressed as 8 declarative equations, written once across the sub-protocols, with the verification code generated automatically.

Metric	Manual	Modular
Equation-construction lines in `sign()`	85	8 declarative eqs.
Equation-construction lines in `verify()`	84	0 (auto-generated)
Hand-written `Equation`/`AMap` objects	60	0
Times each equation is described	2	1

Table 6.2: Groth-Sahai equation-construction effort. The manual approach expresses every equation twice—once for proving and once for verification—whereas the modular approach derives both from a single declarative source.

6.2.4 Summary

The ABS-UCL case study shows that GSkit scales to a non-trivial scheme composed of several interacting sub-protocols. The decisive result is not a reduction in total lines of code—by that measure the modular version is in fact slightly larger—but the elimination of the duplicated, manually maintained equation code that characterizes the direct use of a low-level Groth-Sahai library. The modular implementation replaces 169 lines of imperative equation construction, expressed twice and built from 60 explicit equation and pairing-map objects, with a single declarative description of each sub-protocol's verification, from which both the proving and verification code are generated. This relocation of the equations into one authoritative source is what lowers the error surface of the implementation, since the prover's and the verifier's view of the statement can no longer drift apart, and it is the property that makes a scheme of this complexity practical to build and maintain. These observations directly support the contribution claimed in this thesis and are revisited in the concluding chapter.

Chapter 7

Conclusions

This thesis set out to address a concrete obstacle to the wider adoption of Groth-Sahai proofs: their construction is manual, low-level, and tightly coupled to the underlying pairing primitives, which makes implementations verbose and error-prone. We proposed that a Domain-Specific Language tailored to Groth-Sahai proofs, together with a supporting framework, could abstract these details and let developers express proofs declaratively without sacrificing correctness. The work presented in the preceding chapters supports this hypothesis.

We developed GSkit, a DSL embedded in Python together with a framework that compiles high-level proof specifications into executable proof-generation and verification code. The DSL lets a developer declare the variables, constants, and equations of a proof, while the framework handles commitments, pairing operations, and the construction of the proof elements. Through a structured grammar and type checking, specifications are validated as well-formed before any proof is built, moving a class of errors from runtime to specification time.

We evaluated the framework on two case studies of increasing complexity. The BLS signature scheme served to show that a single pairing-product equation can be extracted directly from a verification routine and turned into a zero-knowledge proof of signature knowledge with minimal developer effort. The ABS-UCL scheme—attribute-based signatures with user-controlled linkability—exercised the framework on a realistic, multi-equation statement combining an MSP policy, a linking token, randomized attribute signatures, and a pseudo-attribute, composed from several independent sub-protocols.

The ABS-UCL case study is the central evidence for our claims, because it let us compare the modular GSkit implementation against an earlier, manual implementation of the same scheme built directly on a low-level Groth-Sahai library. The comparison is qualitative as well as quantitative. In the manual implementation, every Groth-Sahai equation is assembled by hand inside the signing routine and then assembled *again*, almost identically, inside the verification routine—roughly two hundred lines of equation-construction code in which the same relationships are expressed twice, using explicit equation and pairing-map objects. This duplication is precisely where mistakes are easy to introduce and hard to detect, since a discrepancy between the prover's and the verifier's view of an equation does not necessarily

surface as an obvious failure.

The modular GSkit implementation removes this duplication. Each sub-protocol declares its Groth-Sahai verification logic *once*, as a declarative specification attached to its verification method, and the top-level scheme merely *composes* these specifications over a shared proof context. The prover and verifier code are derived from the same single source, so the two can no longer drift apart. Beyond the raw reduction in hand-written equation code, this is the more meaningful result: the framework relocates the equations from imperative, duplicated boilerplate into a single declarative description, which is the property that actually lowers the risk of error and the cost of maintenance.

These results confirm the hypothesis that a high-level DSL can lower the barrier to constructing Groth-Sahai proofs. By raising the level of abstraction from individual equation and pairing-map objects to declarative proof specifications, GSkit makes it feasible to express and maintain a non-trivial scheme such as ABS-UCL with substantially less code and a single, authoritative description of each proof statement.

7.1 Future Work

Several directions remain open for future work. The DSL currently targets the four standard Groth-Sahai equation types; extending it toward batch verification and proof-size optimizations would further increase its practical value. Broadening the library of worked examples beyond BLS and ABS-UCL would help characterize the expressiveness of the language across a wider range of pairing-based protocols, and would provide additional reference implementations for developers adopting the framework. Finally, integrating GSkit into a complete privacy-preserving application—such as the anonymous attribute-based forum that motivated this work—would demonstrate its value end to end, from high-level specification to a deployed system.

In summary, this work shows that a declarative DSL and accompanying framework can express Groth-Sahai proofs—including a non-trivial, multi-equation scheme such as ABS-UCL—at a substantially higher level of abstraction than direct use of a low-level proof library, eliminating the duplicated, manually maintained equation code that characterizes existing implementations and thereby reducing both the effort and the error surface of building these proofs.

Bibliography

[1] Carmine Abate, Bruno Blanchet, Cédric Fournet, Benjamin Grégoire, Nadim Kobeissi, Adrien Koutsos, and Pierre-Yves Strub. Ssprove: A foundational framework for modular cryptographic proofs in coq. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–15. IEEE, 2021.

[2] Joseph A Akinyele, Christina Garman, Ian Miers, Matthew W Pagano, Michael Rushanan, Matthew Green, and Aviel D Rubin. Charm: a framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering*, 3:111–128, 2013.

[3] Aleo Systems Inc. Leo: A programming language for formally verified, zero-knowledge applications. `https://leo-lang.org/`, 2021.

[4] Diego F. Aranha and Conrado P. L. Gouvêa. Relic: An efficient library for cryptography. `https://github.com/relic-toolkit/relic`, 2014.

[5] arkworks contributors. arkworks: An ecosystem for developing and programming with zksnarks. `https://arkworks.rs/`, 2022.

[6] Aztec Labs. Noir: A domain specific language for snark proving systems. `https://noir-lang.org/`, 2023.

[7] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: Computer-aided cryptographic proofs. `https://github.com/EasyCrypt/easycrypt`, 2013.

[8] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Certicrypt: A framework for computational game-based cryptographic proofs in coq. In *International Conference on Interactive Theorem Proving*, pages 74–89. Springer, 2009.

[9] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. Circom: A robust and scalable language for building complex zero-knowledge circuits. `https://github.com/iden3/circom`, 2022.

[10] Karthikeyan Bhargavan, Franziskus Kiefer, and Pierre-Yves Strub. hacspec: Towards verifiable crypto standards. In *International Conference on Research in Security Standardisation*, pages 1–20. Springer, 2018.

[11] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *International conference on the theory and applications of cryptographic techniques*, pages 56–73. Springer, 2004.

[12] COSIC Research Group, KU Leuven. Groth-sahai proofs: Zero to hero! https://www.esat.kuleuven.be/cosic/blog/groth_sahai_proofs-zero_to_hero/, 2022.

[13] Jacob Eberhardt and Stefan Tai. Zokrates - scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings)*, pages 1084–1091. IEEE, 2018.

[14] Ali El Kaafarani, Liqun Chen, Essam Ghadafi, and James Davenport. Attribute-based signatures with user-controlled linkability. In *Cryptology and Network Security: 13th International Conference, CANS 2014, Heraklion, Crete, Greece, October 22-24, 2014. Proceedings 13*, pages 256–269. Springer, 2014.

[15] Ali El Kaafarani and Essam Ghadafi. Attribute-based signatures with user-controlled linkability without random oracles. In *Cryptography and Coding: 16th IMA International Conference, IMACC 2017, Oxford, UK, December 12-14, 2017, Proceedings 16*, pages 161–184. Springer, 2017.

[16] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques (CRYPTO 1986)*, pages 186–194. Springer, 1986.

[17] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In *Advances in Cryptology–EUROCRYPT 2008: 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings 27*, pages 415–432. Springer, 2008.

[18] Vincent Laporte. Verifying cryptographic implementations with jasmin & easycrypt. https://members.loria.fr/VLaporte/files/stv2023_jasmin_easycrypt.pdf, 2023.

[19] Ben Lynn. Pbc library: The pairing-based cryptography library. https://crypto.stanford.edu/pbc/, 2006.

[20] Hemanta K Maji, Manoj Prabhakaran, and Mike Rosulek. Attribute-based signatures. In *Cryptographers' track at the RSA conference*, pages 376–392. Springer, 2011.

[21] QED-it. zkinterface: A standard for zero-knowledge interoperability. https://github.com/QED-it/zkinterface, 2019.

[22] StarkWare Industries. Cairo: A turing-complete stark-friendly cpu architecture. https://cairo-lang.org/, 2021.

[23] Gijs van Lith. groth-sahai: Java implementation of groth-sahai nizk protocol. https://github.com/gijsvl/groth-sahai, 2018.

[24] M. Volkhov. groth-sahai-python: Python reference implementation of groth-sahai proofs. https://github.com/volhovm/groth-sahai-python, 2020.

[25] J.D. White. groth-sahai-rs: A rust library for groth-sahai proofs. https://github.com/jdwhite48/groth-sahai-rs, 2021.

[26] Mariia Zhvanko. How to zk: Noir vs circom. https://medium.com/distributed-lab/how-to-zk-noir-vs-circom, 2024.