

UC Santa Barbara

UC Santa Barbara Electronic Theses and Dissertations

Title

Language and Compiler Support for Correct and Efficient Zero-Knowledge Circuits

Permalink

<https://escholarship.org/uc/item/2cq383hw>

ISBN

9798186382522

Author

Liu, Junrui

Publication Date

2026-06-12

Peer reviewed|Thesis/dissertation

University of California
Santa Barbara

Language and Compiler Support for Correct and Efficient Zero-Knowledge Circuits

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Junrui Liu

Committee in charge:

Professor Yu Feng, Chair
Professor Tevfik Bultan
Professor Ranjit Jhala, University of California, San Diego

June 2026

The Dissertation of Junrui Liu is approved.

Professor Tefvik Bultan

Professor Ranjit Jhala, University of California, San Diego

Professor Yu Feng, Committee Chair

June 2026

Language and Compiler Support for Correct and Efficient Zero-Knowledge Circuits

Copyright © 2026

by

Junrui Liu

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Yu Feng, for his invaluable guidance and hands-on support throughout my PhD journey, which have helped me gradually develop my expertise and “taste” in programming languages (PL) and formal methods research. His feedback has greatly shaped the direction of the present dissertation, and his gentle “pushes” to step out of my comfort zone have helped me grow not only as a researcher but also as a person. Outside of research, I am grateful for the freedom and trust he has given me to pursue my concurrent interests in teaching.

I thank my other committee members, Professor Tevfik Bultan and Professor Ranjit Jhala, for their time and helpful feedback on my research and this dissertation.

Next, I would like to thank my external mentors, Professor Işıl Dillig and Professor Rastislav Bodik. Both of them have enlightened me with fresh yet holistic perspectives on the problems I was working on during our respective collaborations. In days when I felt lost in the weeds of details, I would often recall their wisdom and remind myself to step back and look at the big picture.

My labmates and collaborators have also been a great source of support and inspiration. I am especially grateful to Yanju Chen, who “hand-held” me through the first year of my PhD and taught me the full life-cycle of research projects in PL and all the know-how. I also thank my collaborators on past projects, including Hanzhi Liu, Hongbo Wen, Jiaxin Song, Yanning Chen, Bryan Tan, and many others. Special thanks to Professor Anders Miltner for his invaluable mentorship during the Coda project.

Going further back in time, I am grateful to Professor Rui Meireles and Professor Jonathan Gordon for giving me the first taste of research during my undergraduate years at Vassar College. I was also very fortunate to have met during my last semester at Vassar Professor Anna Gommerstadt, whose OCaml-based Compilers course made me

fall in love with functional programming and PL. She inspired me to pursue a PhD in this field, and eventually a career in teaching and scholarship.

Last but not least, I would like to thank my family and friends for their unconditional love and support. No words come close to capture my gratitude to them, but I hope they know how much I appreciate their presence in my life.

Curriculum Vitæ

Junrui Liu

Education

2021–Present Ph.D. in Computer Science, University of California, Santa Barbara.
2020–2021 M.S. in Computer Science, Yale University.
2016–2020 B.A. in Computer Science, Vassar College.

Publications

Tabby: A Synthesis-Aided Compiler for High-Performance Zero-Knowledge Proof Circuits.

Junrui Liu, Jiaxin Song, Yanning Chen, Hanzhi Liu, Hongbo Wen, Luke Pearson, Yanju Chen, and Yu Feng.

Proceedings of the ACM on Programming Languages, Vol. 9, OOPSLA2, Article 332, August 2025. <https://doi.org/10.1145/3763110>

Refinement Types for Visualization.

Junrui Liu, Jingtao Xia, Nicholas Brown, Yanju Chen, and Yu Feng.

Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), pp. 1871–1881. <https://doi.org/10.1145/3691620.3695550>

Certifying Zero-Knowledge Circuits with Refinement Types.

Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig, and Yu Feng.

Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP '24), pp. 1741–1759. <https://doi.org/10.1109/SP54263.2024.00078>

Conflict-Driven Synthesis for Layout Engines.

Junrui Liu, Yanju Chen, Eric Atkinson, Yu Feng, and Rastislav Bodík.

Proceedings of the ACM on Programming Languages, Vol. 7, PLDI, Article 132, June 2023. <https://doi.org/10.1145/3591246>

Learning Contract Invariants Using Reinforcement Learning.

Junrui Liu, Yanju Chen, Bryan Tan, Işıl Dillig, and Yu Feng.

Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), Article 63, pp. 1–11. <https://doi.org/10.1145/3551349.3556962>

Tree Traversal Synthesis Using Domain-Specific Symbolic Compilation.

Yanju Chen, Junrui Liu, Yu Feng, and Rastislav Bodík.

Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), pp. 1030–1042. <https://doi.org/10.1145/3503222.3507751>

Abstract

Language and Compiler Support for Correct and Efficient Zero-Knowledge Circuits

by

Junrui Liu

Zero-knowledge (ZK) proof systems let one party convince another that a computation was performed correctly while revealing nothing about its secret inputs, and they have rapidly become foundational infrastructure for privacy-preserving and blockchain applications. In nearly all such systems, the computation to be proven must first be expressed as an *arithmetic circuit*: a system of polynomial constraints over a large finite field. Writing these circuits by hand is both error-prone, and performance-critical, since the size and structure of a circuit directly determine the cost of proving and/or verification. Existing domain-specific languages ease the programming burden but address neither problem fully: they give developers no way to specify and verify what a circuit is meant to compute, and their compilers often leave substantial efficiency unexploited.

This dissertation argues that both the correctness and the efficiency of ZK circuits can be addressed systematically through programming-language and compiler techniques, and develops three systems that together substantiate this claim.

CODA is a functional language whose refinement-type system lets developers specify functional-correctness properties alongside their recurring proof patterns. It discharges verification conditions generated by the type checker in a semi-automated way with help of the Rocq proof assistant. Using CODA, we re-implemented widely-deployed circuits from nine projects, and uncovered six previously unknown vulnerabilities.

TESSEL addresses efficiency by introducing a tabular intermediate representation that formalizes the layout-optimization problem for PLONKish circuits, together with a

heuristic layout algorithm that improves over mainstream compilers in practice in many cases.

Finally, TABBY overcomes the limitations of eager heuristics by recasting the same optimization problem as program synthesis, searching the space of equivalent circuits using domain-specific gadgets, goal-directed search, and problem decomposition. Across a suite of real-world benchmarks, TABBY consistently reduces proving time over mainstream compilers.

Together, these systems demonstrate that the source language, the intermediate representation, and the optimization procedure can be co-designed to make ZK circuits both verifiably correct and highly efficient.

Contents

Curriculum Vitae	vi
Abstract	vii
1 Introduction	1
1.1 Motivation	1
1.2 Programming ZK Applications	2
1.3 Challenges and Our Approach	5
1.4 A Note on Terminology	9
1.5 Permissions and Attributions	10
2 Certifying Zero-Knowledge Circuits with Refinement Types	11
2.1 Introduction	11
2.2 Overview	13
2.3 The CODA Language	20
2.4 The CODA Compiler	37
2.5 Implementation	39
2.6 Evaluation	41
2.7 Related Work	47
2.8 Conclusion	50
3 Tessel: An Optimizing Compiler for Efficient Zero-Knowledge Circuits	52
3.1 Introduction	52
3.2 Background and Overview of TESSEL	54
3.3 Preprocessing ZK Programs for Layout Optimization	58
3.4 Layout-Aware Circuit Optimization	63
3.5 Implementation	71
3.6 Evaluation	72
3.7 Related Work	82
3.8 Conclusion	84

4	Tabby: A Synthesis-Aided ZKP Circuit Compiler	85
4.1	Introduction	85
4.2	Background and Overview	88
4.3	Problem Formulation	99
4.4	Circuit Synthesis Guided by Abstract Semantics	108
4.5	Implementation	115
4.6	Evaluation	116
4.7	Related Work	127
4.8	Conclusion	130
5	Conclusion	131
A	Supplementary Material for Chapter 2	133
A.1	Operational Semantics	133
A.2	Metatheory	136
	Bibliography	144

Chapter 1

Introduction

1.1 Motivation

Zero-knowledge (ZK) proof systems, conceived in the 1980s by Goldreich, Micali, and Rackoff [1], have emerged as a promising foundation for privacy-sensitive applications. They allow users to prove that they *know* a secret without revealing the secret itself, often with *succinct* proofs that can be verified quickly. For this reason, ZK proof systems have found numerous applications in blockchain and other privacy-preserving technologies. Privacy-centric digital currencies like ZCash [2] utilize zero-knowledge proofs, as do layer-2 blockchain scaling solutions known as ZK-rollups [3, 4, 5]. More recently, ZK proofs have been adopted as infrastructure across domains spanning privacy-preserving authentication [6], age assurance [7], and machine learning [8].

Fundamentally, a zero-knowledge proof is a cryptographic protocol that involves two parties, a *prover* and a *verifier*. Shared between the two parties is a predicate $R(I, W)$ over variables I and W that encodes the computation the prover claims to know. For example, R can be a predicate that checks the validity of a transaction, the correctness of a machine learning inference, or the satisfaction of an age requirement. The variable I is

called the *public input*, while W is called the *witness* and is known only to the prover. The goal of the prover is to convince the verifier that they know a witness W such that $R(I, W)$ holds, without revealing any information about W itself.¹ For instance, take $R(x, h)$ to be the predicate over witness x and public value h that asserts $\text{SHA256}(x) = h$. That is, h is the SHA256 hash of x . For some public hash h , a prover who knows the secret preimage x can generate a ZK proof that convinces the verifier of this fact without revealing x . The zero-knowledge property ensures that it is computationally intractable for the verifier to learn the exact value of secret witness x from public h and the proof.

Underlying these deployments are two broad families of zero-knowledge proof systems. *zk-SNARKs* encode the relation R as a polynomial constraint system over a finite field; the family includes Groth16 [9], PLONK [10], and PLONK’s many variants such as HyperPLONK [11], TurboPLONK [12], and UltraPLONK [13], which are collectively referred to as *PLONKish* arithmetizations. *zk-STARKs* [14] form the other family and underlie systems such as Cairo [15]; their algebraic intermediate representation (AIR) can be viewed as a restricted form of the PLONKish arithmetization. Despite their cryptographic differences, both families share the same fundamental developer-facing object: the *arithmetic circuit*.

1.2 Programming ZK Applications

In most ZK proof systems, the relation R is encoded using *arithmetic circuits*, which represent systems of polynomial constraints over a fixed finite field $\mathbb{F}_p$ for some prime p .

¹Logically, one can think of the prover as trying to prove the statement $\exists W. R(I, W)$. Normally, to produce a (constructive) proof of an existential statement, one would need to 1) know the existential witness and 2) explicitly embed the witness in the proof so that the proof can be verified. With a ZK proof system, however, the second requirement is dropped: the prover still needs to know the existence of such a W , but the proof system obfuscates the exact value of W while still allowing the verifier to check the validity of the proof. In doing so, the verifier cannot reverse-engineer any information about W from the proof, other than the fact that the prover must have known what W is.

Unfortunately, manually constructing such arithmetic circuits is extremely difficult and error-prone. To see this, consider the following one-liner function that checks whether the input x is zero:

```
1 def isZero(x):
2     return x == 0
```

Surprisingly, to efficiently encode this computation as a circuit, one needs to have the insight to divine an auxiliary witness variable z . Then, the function can be encoded as a circuit consisting of two polynomial equations:

$$x \cdot z + y = 1 \tag{1.1}$$

$$x \cdot y = 0 \tag{1.2}$$

It might not be immediately clear what the role of z is, much less why the above circuit correctly encodes the `isZero` function.² Real-world ZK applications are much more complex than the above example, and the corresponding arithmetic circuits can involve millions of equations [16, 17]. To exacerbate the issue, a single missing equation can cause the entire circuit to be incorrect. For example, the circuit above is a foundational building block for more complex circuits, such as ones that prove the statement “the prover knows two secret values that are equal.” However, if the developer forgets to include Equation 1.2, a malicious prover can generate a bogus proof of the statement “2 is equal to 3.” But because the proof system completely hides the exact value of the secret witnesses, the verifier simply sees a valid proof of “ $\star$ is equal to $\star$ ” and cannot detect the attack.

This example illustrates the difficulty of writing correct and efficient ZK circuits by hand, and the potential security catastrophes that can arise from incorrect circuit

²In fact, the correctness argument for the above encoding relies on a specific property of finite fields, which is that every non-zero element has a multiplicative inverse in the field.

encodings. Indeed, recent attacks have shown that similar bugs in ZK applications can lead to bogus proofs that the verifier accepts as valid, allowing a malicious prover to, for example, forge signatures or counterfeit coins ([18, 19, 20, 21]). These attacks are particularly problematic because they can be exploited *silently* without leaving any observable trace (a consequence of the zero-knowledge property itself), making traditional defense mechanisms ineffective.

To mitigate the programmability and, in turn, *correctness* issues, a broad ecosystem of domain-specific languages (DSLs) has been designed so that arithmetic circuits can be generated (semi-)automatically from the user-specified computation. These DSLs span a wide spectrum of abstractions: at the low end, circuit-based languages like Circom [22] and Halo2 [23] expose the constraint system directly; mid-level DSLs such as Noir [24], Leo [25], ZoKrates [26], and CirC [27] layer more conventional syntax and familiar programming constructs (variables, loops, modules) on top of explicit constraint programming.³

However, despite these higher levels of abstraction and compiler support, the correctness issue persists across the entire spectrum: programs written in such DSLs still manipulate field equations under the hood and remain prone to the class of errors discussed above, and the compiler itself becomes an additional attack surface. Static analysis and formal verification techniques are therefore essential for the security of ZK applications, and a small body of work has begun to address this gap via lightweight static analysis (e.g., Circomspect [31], Picus [32], ZKAP [33]) or compiler verification (e.g., Leo [25], CCVC [34]). Each of these, however, targets a fixed class of vulnerability or a fixed compiler; none lets developers formally specify and statically check the functional correctness of their own ZK applications, leaving them vulnerable to silent attacks.

An orthogonal issue introduced by the aforementioned DSLs is that abstractions can

³A recent yet orthogonal approach to ZK programming is the development of *zero-knowledge virtual machines* such as RISC Zero [28], SP1 [29], and Jolt [30], which dispense with the source DSL altogether by implementing general-purpose ISAs with bespoke proof systems.

come at the cost of circuit *efficiency*: a compiler for a high-level DSL may generate suboptimal circuits that lead to higher proving and/or verification cost. To circumvent this, developers sometimes revert to manually building the constraint system in low-level circuit-based DSLs like Circom and Halo2 (e.g., [35]), which provide fine-grained control over the resulting arithmetic circuit. However, this approach is not ideal because it requires developers to have deep expertise in both the domain of ZK applications and the underlying arithmetic circuits. Moreover, writing low-level code in a circuit-based DSL is error-prone, exacerbating the correctness issue described above. Compiler-side optimization efforts do exist, but to date have largely focused on conventional compiler optimizations (e.g., Leo [25], Noir [36]) or rewrite-based simplifications driven by hard-coded heuristics (e.g., [16]). None of these systems systematically formalize the optimization space for low-level ZK circuits, let alone search it effectively.

Instead, the present thesis argues that both issues are best tackled not through scarce manual expertise, but through programming-language and compiler techniques. Concretely, we defend the following claim: *the functional correctness of a zero-knowledge circuit can be specified and statically verified within its source language through a refinement-type discipline, while its efficiency can be obtained automatically by a compiler that recasts circuit optimization as a precisely formalized search problem.*

1.3 Challenges and Our Approach

The previous section identified two high-level problems — *correctness* and *efficiency* — that any practical ZK compiler must address. Both goals, however, run into specific obstacles that existing tools have not solved. Below, we describe each challenge and outline our approach; the details follow in the subsequent chapters.

Challenge 1: how can we let developers specify *and* prove the functional correctness of their circuits? The DSLs surveyed above lower the cost of *writing* arithmetic circuits, but they leave the developer with no way to say what the circuit is supposed to compute, let alone prove that it does. The bug-finders described earlier each target a fixed class of vulnerability rather than a developer-supplied specification. What is missing is a single language in which functional correctness specifications can be written alongside the code they describe, in a syntax light enough that developers will actually use it, together with a verification discipline that can confirm those specifications hold.

Our approach (Chapter 2). We design CODA, a DSL in which developers write *refinement-type* annotations [37] directly alongside their circuit code to specify full functional-correctness properties. Intuitively, the refinement type specifications in CODA make it possible to express the correspondence between the *computation* that the programmer has in mind and the actual *arithmetic circuit* that underlies the zero-knowledge proof system. As a result, refinement types serve as a bridge between the developers’ mental computational model and the mathematical objects that are used to realize that computation in a zero-knowledge way.

A side challenge that arises is that mainstream refinement-type systems [37, 38, 39, 40, 41] discharge quantifier-free typing obligations to an SMT solver. This is inadequate for ZK programs, however, because proving full functional-correctness properties in this domain requires complex reasoning about the theory of finite fields combined with other theories such as non-linear integer arithmetic and quantified reasoning, a task for which off-the-shelf SMT solvers offer no useful decision procedure. Although ZK-specific solvers [42, 43, 44, 32, 45, 46] make progress on restricted fragments, a general-purpose verification tool is still out of reach.

To bridge this gap, the CODA type checker emits proof goals in the Rocq/Coq proof assistant [47], paired with a tactic library that codifies recurring patterns that arise

when certifying ZK applications. The resulting workflow sits between fully automated verification and fully manual proof: the type system enforces the bookkeeping and decomposes a large verification task into independent subgoals, which the developer can discharge with domain-specific lemmas. Using CODA, we re-implement 77 widely-deployed Circom circuits from 9 projects, certifying 66 of them in full and uncovering 6 previously unknown vulnerabilities.

Challenge 2: how to formalize and search the optimization space for low-level ZK circuits? Once a circuit is verified correct, the question becomes how to make it cheap. As the previous section observed, existing compiler-side optimization efforts either lift conventional passes from general-purpose languages or apply rewrite-based simplifications driven by hard-coded heuristics. They leave as a gap a precise specification of the optimization space for ZK circuits, which is a prerequisite for any principled investigation of how to search it efficiently. The issue is particularly acute for PLONKish circuits, a class of arithmetizations that are widely used in practice but have a complex optimization space that is not well understood. For example, PLONKish circuits are not “bags of constraints” but have a rich tabular structure where cells hold variables and inter-cell relationships are expressed via *custom gates*, which are polynomial constraints that can be custom-defined for specific use cases. Optimizing such circuits requires making interconnected layout decisions about how the computation is distributed across the table.

Our approach (Chapter 3 and Chapter 4). We attack the two halves of this challenge in two chapters. TESSEL (Chapter 3) addresses the formalization half by introducing a tabular intermediate representation (IR) that exposes the yet-to-be-determined layout of a low-level circuit explicitly, giving us a vocabulary in which the optimization problem can be stated precisely. The chapter describes a compilation pipeline that partially evaluates the source DSL program down to the tabular IR, and then applies an eager,

heuristic-based layout algorithm that can yield measurable improvements over mainstream compilers on many workloads in practice. Then, TABBY (Chapter 4) addresses several limitations of the heuristic algorithm. There, we reformulate the optimization as a *program synthesis* problem [48, 49, 50, 51]. TABBY identifies interesting subspaces of the optimization problem and, within each, exhaustively searches for all equivalent circuits rather than greedily traversing the space. To further cut down the search space, the synthesis algorithm is guided by decomposition techniques and abstract semantics. Across our benchmark suite, TABBY produces consistent reductions in proving and verification costs over mainstream compiler.

In short, the present thesis makes the following contributions:

- In Chapter 2, we propose a functional DSL, called CODA, for ZK programming. Unlike existing DSLs, CODA allows developers to simultaneously write the application logic and formally specify its correctness within the same language via lightweight refinement-type annotations [37]. Furthermore, CODA comes with a type checker that can semi-automatically *certify* the functional correctness of a ZK application, backed by formal proofs using the Rocq/Coq interactive theorem prover.
- In Chapter 3, we describe a compilation framework that converts high-level ZK programs into low-level arithmetic circuits. Importantly, we formalize the optimization problem for PLONKish arithmetization [10] and propose a heuristic layout algorithm that effectively addresses it in practice.
- In Chapter 4, we present TABBY, a synthesis-aided compiler that overcomes the limitations of eager optimization heuristics by searching the space of equivalent circuits to generate efficient low-level arithmetic circuits from high-level ZK programs.

1.4 A Note on Terminology

The terms “verification” and “proof” carry different technical meanings in the cryptographic and formal methods communities. As the present thesis straddles both communities, it is worth clarifying the difference up front, so that the reader can identify which meaning is intended in each context.

In the formal methods community, a (formal) proof is simply a mathematical proof that certifies the truth of a logical proposition beyond doubt, through a sequence of logical deductions. The prover is assumed to be honest and hides nothing, making both the proposition and the entire sequence of deductions publicly available. Verification merely traces through the same sequence of deductions to confirm that each step is a valid application of a logical inference rule, and that the final conclusion is the proposition being proven. In this setting, the proof is the entire sequence of deductions, and verification is a deterministic process that either succeeds or fails.

In contrast, in the cryptographic community, the act of proving and verifying happens in an interactive proof system [1]. Typically, the prover makes a (computational) statement that may conceal secret information. Furthermore, the prover is assumed to be computationally more powerful than the verifier, and is not assumed to be honest. The verifier engages the prover in a dialogue, posing challenges for the prover to respond to, in an attempt to catch a dishonest prover making a false claim. The process proceeds for a finite number of rounds, until the probability of a dishonest prover passing the verifier’s checks is made arbitrarily small. Thus, the verification process is inherently probabilistic: a dishonest prover may be able to pass the verifier’s checks by sheer luck. The proof is the transcript of the interaction between the prover and verifier, which can be checked by the verifier to determine whether to accept or reject the claim.⁴ In ZK proof systems,

⁴Note that this process can be made, and is typically made, non-interactive by using the Fiat-Shamir heuristic [52], which enables the prover to generate a proof a priori without interacting with the verifier;

the computational statement is typically encoded as an arithmetic circuit, and the proof is a cryptographic object that the proof system generates from the circuit, the public input, and the secret witness.

1.5 Permissions and Attributions

1. The content of Chapter 2 is the result of a collaboration with Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig, and Yu Feng, and has previously appeared as “Certifying Zero-Knowledge Circuits with Refinement Types” in the Proceedings of the 2024 IEEE Symposium on Security and Privacy [53]. © 2024 IEEE. Reprinted, with permission.
2. The content of Chapter 4 is the result of a collaboration with Jiaxin Song, Yanning Chen, Hanzhi Liu, Hongbo Wen, Luke Pearson, Yanju Chen, and Yu Feng, and has previously appeared as “Tabby: A Synthesis-Aided Compiler for High-Performance Zero-Knowledge Proof Circuits” in the Proceedings of the ACM on Programming Languages (OOPSLA 2025) [54].

after this, anyone can act as a verifier by checking the proof. Thus, in future chapters, we often speak of the prover generating a proof and the verifier verifying it as two distinct stages.

Chapter 2

Certifying Zero-Knowledge Circuits with Refinement Types

2.1 Introduction

In this chapter, we examine the *correctness* aspect of ZK programming. Recall that a particularly problematic aspect of bugs in ZK applications is that they are extremely difficult to detect, *even once they are exploited*. Unlike other common types of attacks like denial-of-service or privilege escalation, ZK attacks allow a malicious party to *silently* exploit the system, without leaving any observable trace. Hence, traditional defense mechanisms, such as run-time monitoring or sandboxing, are completely powerless at mitigating attacks caused by bugs in ZK proof system implementations. This leaves *static analysis & verification* as the only viable mechanism for preventing silent attacks on ZK proof systems.

© 2024 IEEE. Reprinted, with permission, from Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig, and Yu Feng, “Certifying Zero-Knowledge Circuits with Refinement Types,” *Proceedings of the IEEE Symposium on Security and Privacy*, 2024.

Motivated by this observation, this chapter presents a new statically-typed language called CODA for building ZK applications. Critically, CODA makes it possible to *formally specify* and *statically check* properties of a ZK application through a rich *refinement type* system. Intuitively, the refinement type specifications in CODA make it possible to express the correspondence between the *computation* that the programmer has in mind and the actual *arithmetic circuit* that underlies the zero-knowledge proof system. As a result, refinement types serve as a bridge between the developers’ mental computational model and the mathematical objects that are used to realize that computation in a zero-knowledge way.

At a high level, CODA serves the same role as a domain-specific language like Circom, in that it is intended to facilitate the construction of arithmetic circuits. However, unlike Circom which is a low-level imperative language, CODA is a functional language with higher-order combinators. Beyond making it possible to express the same computation in a more concise and declarative way, CODA is designed to facilitate static type checking. In particular, given the refinement type specifications annotated by the user, the CODA compiler generates a set of logical formulas whose validity entails the correctness of the circuit. That is, if the formulas generated by the CODA compiler can be proven to be logically valid, this constitutes a proof that the program conforms to its specification. Thus, type checking in CODA essentially amounts to formal verification.

One of the key challenges in formally verifying ZK applications is that they require reasoning about polynomial equations over large prime fields. This fact is important because current automated theorem provers (in particular, SMT solvers) do not provide adequate support for logical reasoning in such a theory. Hence, in contrast to recent work on *logically qualified data types* [37, 38, 39, 40, 41] that discharges typing constraints via an SMT solver, doing so is unfortunately not an option for the CODA compiler. To deal with this challenge, CODA outputs a set of lemmas in Coq [47] that can be proven in

an interactive manner. Because many of the lemmas output by the CODA type checker correspond to complex number-theoretic theorems, it is unrealistic to expect that they can be discharged completely automatically. Thus, while human involvement in the type checking process is unavoidable in some cases, CODA tries to reduce the manual proof burden by providing tactics to codify common proof patterns in this domain.

We have used CODA to re-implement and formally verify 77 complex arithmetic circuits from 9 Circom projects. Out of the 77 circuits under consideration, we were able to use CODA to formally verify 66 circuits and uncovered 6 previously unknown vulnerabilities in popular Circom projects.

To summarize, this chapter makes the following key contributions:

- We introduce CODA, a statically-typed functional DSL for facilitating the construction of arithmetic circuits and their compilation down to R1CS constraints [55].
- We describe a refinement type language for specifying functional correctness properties of ZK applications and present type checking rules that are formally proven to be sound. In other words, we show that type checking in CODA corresponds to formal verification.
- We re-implement 77 widely-used ZK circuits from popular Circom projects in CODA and show that CODA is useful for specifying and formally verifying these circuits. Furthermore, our verification effort uncovered previously unknown vulnerabilities.

2.2 Overview

In this section, we motivate our proposed approach with the aid of a motivating example. To this end, we first present an existing circuit written in Circom and then explain the CODA workflow.

```

1  template BigLessThan(k){
2      signal input a[k];
3      signal input b[k];
4      signal output out;
5      component lt[k];
6      component eq[k];
7      for (var i = 0; i < k; i++) {
8          lt[i] = LessThan();
9          lt[i].in[0] === a[i];
10         lt[i].in[1] === b[i];
11         eq[i] = IsEqual();
12         eq[i].in[0] === a[i];
13         eq[i].in[1] === b[i];
14     }
15     component ors[k - 1];
16     component ands[k - 1];
17     component eq_andands[k - 1];
18     for (var i = 0; i <= k - 2; i++) {
19         ands[i] = AND();
20         eq_andands[i] = AND();
21         ors[i] = OR();
22         if (i == 0) {
23             ands[i].a === eq[1].out;
24             ands[i].b === lt[0].out;
25             eq_andands[i].a === eq[1].out;
26             eq_andands[i].b === eq[0].out;
27             ors[i].a === lt[1].out;
28             ors[i].b === ands[i].out;
29         } else {
30             // Bug! The next line should be:
31             // ands[i].a === eq_andands[i-1].out
32             ands[i].a === ands[i-1].out;
33             ands[i].b === lt[i].out;
34             eq_andands[i].a === eq_andands[i-1].out;
35             eq_andands[i].b === eq[i].out;
36             ors[i].a === ors[i-1].out;
37             ors[i].b === ands[i].out;
38         }
39     }
40     out === ors[k-1].out;
41 }
42

```

Figure 2.1: A Circom program for expressing the less-than relation between a and b . The `out` signal is expected to be a binary field element that is equal to 1 if and only iff a represents a smaller integer than b .

2.2.1 Motivating Example

Figure 2.1 shows a function called **BigLessThan** implemented in Circom. This function takes two arrays **a** and **b** representing two integers in Big Endian notation. For a specific value of **k** (representing the number of bits), the Circom compiler generates an arithmetic circuit (i.e., a polynomial equation over a finite field) such that the (binary) output signal called **out** is 1 iff the integer represented by **a** is less than that represented by **b**, and 0 otherwise. While *computing* the correct value of **out** may not seem particularly challenging, what makes this task tricky is that we actually need to *generate a set of constraints* over the signals **out**, **a**, and **b** satisfying a certain property. If the generated set of constraints is incorrect, a malicious party can engineer bogus proofs that are accepted by the verifier [56].

For the purposes of this chapter, it is not necessary to understand the Circom code shown in Figure 2.1 except to realize that the Circom code is actually quite involved despite being based on the following simple observations:

- **Observation 1:** The length- i prefix of **a** is less than the length- i prefix of **b** if the length- $(i-1)$ prefix of **a** is less than the length- $(i-1)$ prefix of **b**, or if the length- $(i-1)$ prefixes are equal but $a[i] < b[i]$.
- **Observation 2:** The length- i prefix of **a** is equal to the length- i prefix of **b** if the length- $(i-1)$ prefixes are equal and $a[i] = b[i]$.

Unfortunately, the actual Circom code generating the constraints (using the `===` operator) is quite non-trivial and uses the **lt** and **eq** *components* (which are similar to library functions) to help implement this logic. While understanding the exact details of the Circom code is not necessary for this chapter, the key take-away is that the code is far from trivial, and, in fact, contains a subtle bug at line 32, with the appropriate fix indicated in comments immediately above. Such a bug can allow a malicious prover to

```

1 circuit BigLessThan
2   (k: {Z | 0 <= v})
3   (a: {F | binary v}^k)
4   (b: {F | binary v}^k)
5   -> {F | v = (|a| < |b|)} {
6     let (lt, _) = iter 0 (k-1) (
7       \i. \ (lt, eq). (
8         #Or lt (#And eq (#LessThan a[i] b[i])),
9         #And eq (#Eq a[i] b[i]))
10      )
11      (0, 1)
12      inv: (\i.
13        {F | v = (|a[:i]| < |b[:i]|)} *
14        {F | v = (|a[:i]| = |b[:i]|)})
15      in lt
16  }
17

```

Figure 2.2: A CODA DSL program for the same function in Figure 2.1.

construct the input arrays **a** and **b** in such a way that the generated constraints will be satisfied even though the integer represented by **a** is larger than that represented by **b**.

The bug in this example may look relatively innocuous, but the consequences of such a mistake can be quite catastrophic in the context of a ZK application. For example, if this circuit was used in a DeFi application to ensure that parties cannot transfer more money than they have, an attacker could exploit this bug to transfer money than they actually own. Furthermore, in such a privacy-centric DeFi application based on a ZK proof system, it would be very difficult to detect that something bad is happening: since no one except the attacker knows exactly how much money they own, no party can detect that money is being stolen if the verifier itself is buggy.

2.2.2 Our approach

Figure 2.2 shows the implementation of the **BigLessThan** function in CODA. Even though the CODA code in Figure 2.2 compiles down to the same R1CS constraint representation as the Circom code from Figure 2.1, the CODA program is much more

concise and declarative and allows the user to directly implement **Observation 1** and **Observation 2** without being bogged down in low-level, orthogonal details. In particular, the implementation of **BigLessThan** in CODA uses the functional **iter** construct of the form:

$$\text{iter}_{\tau} \ e_s \ e_e \ e_f \ e_a$$

where e_s, e_e denote the start and end indices of the iterator, e_f is a lambda abstraction that is repeatedly applied to the accumulator, and e_a is the initial value of the accumulator. Hence, the code in Figure 2.2 iterates from 0 to $k - 1$ (inclusive) and repeatedly applies e_f to the accumulator, which is initialized to the value $(0, 1)$. In this case, the accumulator is a pair of booleans (**lt**, **eq**) where **lt** (resp. **eq**) indicates whether the length i prefix of a is less than (resp. equal to) that of b . Observe that the new value of **lt** is calculated as:

$$lt \vee (eq \wedge (a[i] < b[i]))$$

which corresponds directly to **Observation 1** from earlier. Similarly, the computation of **eq** also directly follows **Observation 2**. Hence, the CODA implementation of **BigLessThan** allows the developer to implement the core logic in a relatively straightforward way by *composing* existing circuits that implement boolean disjunction (**#Or**), conjunction (**#And**), and field element comparison (**#LessThan**).

Specifying circuit behavior. Beyond making it significantly easier to construct arithmetic circuits, a key aspect of the CODA DSL is that it allows *specifying* important properties of the circuit as refinement types, which are indicated in blue in Figure 2.2. As we can see from the type annotation on the input **k**, refinement types have the form $\{T \mid \phi\}$ where T is a base type and ϕ is a *logical qualifier*. In this case, the base type of **k** is an integer, denoted as **Z**, and the logical qualifier $0 \leq \nu$ further states that the value

of $\mathbf{k}$ is non-negative. The type annotations on $\mathbf{a}$ and $\mathbf{b}$ are of the form τ^k , indicating that they are arrays of size k with elements of type τ . In this case, the element types of $\mathbf{a}$ and $\mathbf{b}$ are field elements (indicated as $\mathbf{F}$), and the logical qualifier further restricts their values to be binary.

The most interesting part of the specification is the type annotation for the return value of **BigLessThan**, which has the following refinement type:

$$\{\mathbf{F} \mid \nu = (|a| < |b|)\}$$

This states that the return value of the function is a binary field element which is 1 iff the integer represented by $\mathbf{a}$ is less than the one represented by $\mathbf{b}$, where the operator $|x|$ interprets an array as a base-2 big integer. We note that the expression $|a| < |b|$ is valid in CODA's type system, but it is fundamentally *not* a valid *program* expression. In fact, it is impossible to define the custom $|\cdot|$ operator as a CODA function to map an array of field elements to the integer that it represents (as a field element), since the array may represent an integer larger than the size of the finite field.

Specifying invariants. In order to prove that the CODA implementation of **BigLessThan** adheres to its specification, the developer also needs to annotate the **iter** construct with a so-called *loop invariant*, which specifies the refinement type of the accumulator. For this example, the annotation at lines 12-14 specifies that the accumulator is a pair of (binary) field elements, where the first element indicates whether the length- i prefix of $\mathbf{a}$ is less than the length- i prefix of $\mathbf{b}$, and the second pair indicates whether they are equal. Because automated inference of loop invariants is, in general, not always possible, the CODA DSL requires the developer to annotate every usage of the **iter** construct with a refinement type specification of its accumulator and utilizes it in the verification process.

Verification. In order to prove that the implementation of **BigLessThan** conforms to its specification, CODA performs static type checking. In particular, CODA utilizes the refinement type annotations to generate a set of *proof obligations*, also referred to as *lemmas*. If all of these lemmas are proven to hold, this constitutes a formal proof that the implementation conforms to its specification. For our running example, CODA generates a total of 15 lemmas, one of which is the following:

$$\begin{aligned}
 & \forall (k \ i : \text{nat})(a \ b : F \ \text{list}), \\
 & \quad \text{length}(a) = k \wedge \text{length}(b) = k \\
 & \quad \wedge 0 \leq i < k \wedge |a[: i]| < |b[: i]| \\
 & \quad \implies |a[: i + 1]| < |b[: i + 1]|.
 \end{aligned}$$

Essentially, this lemma states that, if the length- i prefix of **a** represents an integer smaller than the length- i prefix of **b**, then the same is true for the length- $(i + 1)$ prefixes. Note that this lemma is needed for proving the correctness of the loop invariant, which in turn is essential for establishing that the return value of **BigLessThan** has the intended refinement type specification.

Discharging proof obligations. Since proving properties about ZK applications often requires proving number-theoretic lemmas, there is unfortunately little hope of discharging the proof obligations generated by the CODA type checker *completely* automatically. Thus, rather than using an automated theorem prover (e.g., SMT solver), the CODA type checker instead outputs Coq theorems [47] to be proven interactively with the aid of CODA’s domain-specific Coq tactics. In our running example, 10 of the lemmas can be discharged fully automatically and are not shown to the user. The user does need to be involved for proving the remaining 5 lemmas; however, CODA’s tactics (as well as existing tactics

provided by Coq and the Fiat crypto library [57]) are still useful for simplifying the proof.

Code generation. Once proof obligations are discharged and type checking succeeds, the CODA compiler generates a Rank-1 Constraint System (R1CS) [55] representation of the corresponding CODA program. We choose to compile CODA code down to R1CS because there are existing zkSNARK generators [58] that can be used to produce a prover and verifier from a set of R1CS constraints. For our running example, the R1CS constraints generated by the CODA compiler are roughly of the same size and quality as the constraints generated by the Circom compiler for the code from Figure 2.1.¹

Threat model. We consider a trustless setup where the attacker has full access to the implementation of the ZK circuit, both at the source code level as well as the R1CS level. We also assume that the attacker has access to both the prover and the verifier generated from this circuit. We further assume that the attacker does not have to use the zero-knowledge proof generated by the prover; instead, they can replace the proof with their own bogus version.

2.3 The CODA Language

In this section, we present the syntax, semantics, types, and compilation procedure of the CODA DSL.

2.3.1 CODA Syntax and Semantics

At a high level, the CODA DSL is designed with three key goals in mind:

- **Programmability:** The DSL should make it possible to concisely and conveniently express arithmetic circuits of practical interest. To achieve this goal, CODA is modular

¹The size of the R1CS constraints is important both for efficiency and also for reducing the time and space costs of producing the proof [59].

$r ::=$	circuit $C \overrightarrow{(x_i : T_i)} \rightarrow T \{e\}$	Circuit declaration
$e ::=$		Expression:
	c	constants
	x	variable
	$\star$	wild card
	$b \mid z \mid f$	arithmetic expressions
	assert $e_1 = e_2$	assertion
	$e_1 \odot e_2$	binary relation
	$\#C \overrightarrow{e_i}$	circuit reference
	a	array operator
	t	product operator
	$\lambda x : \tau. e$	function abstraction
	$e_1 e_2$	function application
	let $x = e_1$ in e_2	variable binding
	iter $_{\lambda i. \tau} e_s e_e e_f e_a$	iteration
$b ::=$	$\neg e \mid e_1 \odot e_2$	Boolean expression
$z ::=$	$-\mathbb{Z}e \mid e_1 \oplus e_2$	Integer expression
$f ::=$	$-e \mid e_1 \otimes e_2$	Field expression
$t ::=$		Product operator:
	$(e_1, \dots, e_n)_\phi$	product constructor
	match e with $(x_1, \dots, x_n) \rightarrow e'$	product destructor
	$e.i$	product indexing
$a ::=$		Array operator:
	$e_1 :: e_2$	array constructor
	map $e_1 e_2$	map
	length e	length
	sum $e_1 e_2 e_3$	sum
	$e_1 [e_2]$	array indexing
$\odot \in$	$\{\wedge, \vee\}$	Boolean operators
$\oplus \in$	$\{+\mathbb{Z}, -\mathbb{Z}, *\mathbb{Z}\}$	Integer operators
$\otimes \in$	$\{+, -, *, /\}$	Field operators
$\odot \in$	$\{=, <, \leq\}$	Binary relations

Figure 2.3: Syntax of CODA Programs

and allows composing different circuits. It also incorporates higher-order combinators that allow mapping over arrays and performing aggregation.

- **Compilability:** Since our ultimate goal is to facilitate the generation of a ZK proof system (consisting of a prover and verifier), it should be possible to compile CODA programs down to R1CS constraints, for which existing techniques can generate ZK proof systems. To achieve this goal, CODA primarily allows computation over finite field elements and only allows using other values like integers and booleans in restricted ways.
- **Verifiability:** To allow CODA programs to be statically verified, CODA is both purely functional and based on a static refinement type system. Furthermore, to allow complex properties to be verified, CODA does not insist on decidable type checking. Instead, it outputs Coq lemmas to be proven semi-automatically using a combination of proof tactics and manual intervention where necessary.

Figure 2.3 shows the syntax for CODA programs. Top-level CODA programs are referred to as *circuits* (parametrized over a finite field $\mathbb{F}$) and consist of three components: (1) $\overline{x_i : T_i}$, which is a sequence of input parameters x_i and their corresponding type T_i , (2) T , the output type of the circuit, and (3) e , the body of the circuit. Given some choice of values for $\overline{x_i}$, a CODA program evaluates the body expression e and returns the evaluation result v . At a high level, CODA is a functional programming language with domain specific constructs for building ZK circuits as well as several standard functional combinators. In what follows, we explain the key features of the CODA language.

Standard functional constructs in CODA include integer and boolean expressions, arrays and tuples, lambda abstractions, function applications, iterators, and variable bindings. A variable binding `let  $x = e_1$  in  $e_2$` binds the result of evaluating expression e_1 to the (immutable) variable x and evaluates e_2 . The expression `iter $_{\lambda i.\tau}$   $e_s$   $e_e$   $e_f$   $e_a$` performs aggregation over indices. Specifically, e_s, e_e are start and end indices respectively, e_f is a function, and e_a is the initial value of the accumulator. Starting with the initial value e_a ,

the function e_f is applied on the accumulator to produce a new value, and this process continues for all indices in the range $[e_s, e_e)$. Note that this construct has a subscript $\lambda i.\tau$ indicating the *loop invariant*, which is used in type-checking, as discussed in Section 2.3.3.

Beyond the standard functional constructs, CODA allows expressing computations over finite fields, which are crucial for building ZK circuits. Finite field expressions in CODA include field addition, subtraction, and multiplication. These operations are similar to their integer counterparts except that they occur modulo the field size $\mathbb{F}$ and may therefore over- or under-flow where their corresponding integer counterparts do not.

As CODA programs need to encode *relations* (rather than *mathematical functions*), they allow non-determinism, which is expressed via the $\star$ construct in the DSL, which can evaluate to *any* field element. Thus, a given CODA program can have multiple outputs even when executed on the same input. For example, a CODA function that takes as input a value x and “returns” $\star$ corresponds to a relation R that evaluates to true for any pair (x, y) (where y is the “output” of the CODA function).

In addition to allowing completely non-deterministic choices, CODA also allows constraining non-determinism via *assertions*. For example, consider the following CODA circuit:

```
circuit C (a : F) : F =
    let b =  $\star$  in
    let _ = assert a * b = 0 in b
```

This circuit essentially corresponds to a relation $C(a, b)$ that evaluates to true if and only if $a * b = 0$ for some field elements a, b . Note that assertions are restricted to predicates of the form $e_1 = e_2$, as the resulting constraint must be compilable down to R1CS, which may not be feasible for arbitrary predicates.

We conclude our discussion of CODA syntax with circuit references of the form $\#C(x)$.

At a high-level, circuit references allow CODA programs to refer to other circuits in a *compositional* way. Operationally, they allow developers to pretend that CODA circuits can be called like regular functions (even though they are not). For instance, consider a CODA program that returns $\#C(x)$, where C is some circuit representing a relation $R(x, y)$. Then, the top-level CODA circuit C' can be viewed as syntactic sugar for the following program:

```
circuit C' (a : F) : F =
    let b = * in
    let _ = assert R(a, b) in b
```

2.3.2 CODA Semantics

The semantics of CODA require *valuations*, mappings from variables to values. We formalize the semantics of CODA expressions using judgments of the form $\sigma \vdash e \Downarrow v$, indicating that e can evaluate to value v under valuation σ . As emphasized earlier, CODA programs denote relations rather than mathematical functions; hence, it is possible for an expression e to evaluate to two distinct values v, v' under the same valuation.

While we present the semantics of the full CODA language in Appendix A.1, Figure 2.4 shows the semantics for a representative subset of CODA expressions. The semantics utilize an environment Δ mapping circuits to their definitions in the E-CREF rule, and the iterator semantics are defined recursively. Finally, note that, for **assert** statements, evaluation gets stuck for field elements that do not satisfy the predicate. Thus, if we think of a CODA circuit C with inputs $\bar{x}$ and body e as a relation $R_C((x_1, \dots, x_n), y)$, we have $R_C((in_1, \dots, in_n), out)$ evaluates to true if and only if $(x_1 \mapsto in_1, \dots, x_n \mapsto in_n) \vdash e \Downarrow out$.

$v ::=$	Values:	$\sigma ::=$	Valuation:
c	constants	$\emptyset$	empty valuation
$(v_1, \dots, v_n)$	product value	$\sigma[x \mapsto v]$	augmentation
$v_1 :: v_2$	array value		
$\text{Closure}(\lambda x : \tau. e, \sigma)$	closure		
$\frac{}{\sigma \vdash v \Downarrow v}$ E-VALUE	$\frac{\sigma(x) = v}{\sigma \vdash x \Downarrow v}$ E-VAR	$\frac{v \in \mathbb{F}}{\sigma \vdash \star \Downarrow v}$ E-NONDET	
$\frac{\sigma \vdash e_1 \Downarrow f \quad \sigma \vdash e_2 \Downarrow f \quad f \in \mathbb{F}}{\sigma \vdash \text{assert } e_1 = e_2 \Downarrow \text{unit}} \text{ E-ASSERT}$			
$\frac{\sigma \vdash e_1 \Downarrow f_1 \quad \sigma \vdash e_2 \Downarrow f_2 \quad f_1 \otimes f_2 = f}{\sigma \vdash e_1 \otimes e_2 \Downarrow f} \text{ E-FBINOP}$			
$\frac{}{\sigma \vdash \lambda x : _. e \Downarrow \text{Closure}(\lambda x : _. e, \sigma)} \text{ E-LAM}$			
$\frac{\sigma \vdash e_1 \Downarrow \text{Closure}(\lambda x : _. e, \sigma') \quad \sigma \vdash e_2 \Downarrow v \quad \sigma'[x \mapsto v] \vdash e \Downarrow v'}{\sigma \vdash e_1 e_2 \Downarrow v'} \text{ E-APP}$			
$\frac{\begin{array}{l} \sigma \vdash e_l \Downarrow v_l \text{ for all } l \in \{s, e, f, a\} \\ v_s \in \mathbb{Z} \quad v_e \in \mathbb{Z} \quad v_s \geq v_e \end{array}}{\sigma \vdash \text{iter}_{\lambda i. \tau} e_s e_e e_f e_a \Downarrow v_a} \text{ E-ITER0}$			
$\frac{\begin{array}{l} \sigma \vdash e_l \Downarrow v_l \text{ for all } l \in \{s, e, f, a\} \\ v_s \in \mathbb{Z} \quad v_e \in \mathbb{Z} \quad v_s < v_e \end{array} \quad \sigma \vdash \text{iter}_{\lambda i. \tau} (v_s +_{\mathbb{Z}} 1) v_e v_f (v_f v_a) \Downarrow v}{\sigma \vdash \text{iter}_{\lambda i. \tau} e_s e_e e_f e_a \Downarrow v} \text{ E-ITERS}$			
$\frac{\begin{array}{l} \Delta(C) = \text{circuit } R(x_1 : T_1) \cdots (x_n : T_n) \rightarrow T \{e\} \\ \sigma \vdash e_i \Downarrow v_i \text{ for all } i \in \{1, \dots, n\} \quad \sigma[x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] \vdash e \Downarrow v \end{array}}{\sigma \vdash \#C e_1 \cdots e_n \Downarrow v} \text{ E-CREF}$			

Figure 2.4: Selected Semantics Rules of CODA Expressions

$T ::=$		Basic types:
$\mathbf{F}$		field element
$\mathbf{Int}$		integer
$\mathbf{Bool}$		boolean
$T_1 \times \dots \times T_n$		product (Unit denotes empty)
$[T]$		array type
$\{\nu : T \mid \phi\}$		refinement type
$\tau ::=$	T	
$ $	$x : \tau_1 \rightarrow \tau_2$	Function type
$\phi ::=$		Logical qualifiers:
t		refinement term
$\neg \phi \mid \phi \wedge \phi \mid \phi \vee \phi$		boolean operators
$\forall_{i \in [0, k)} \cdot \phi$		bounded quantification
$t ::=$		Refinement terms:
$\mathbf{true} \mid \mathbf{false}$		boolean constants
F		field size
e		expression
$\mathbf{toN}(t) \mid \mathbf{toZ}(t)$		field to integer conversion

Figure 2.5: Syntax of CODA Types

2.3.3 CODA Type System

CODA facilitates static verification through its rich type system based on *refinement types*. At a high level, refinement types allow developers to express functional correctness requirements on their ZK circuits as type annotations that must be checked by the compiler.

Figure 2.5 shows the syntax for CODA’s refinement type system, which can be split into four core classes, namely, *base types*, *compound types*, *function types*, and *refinement types*. The three base types include **F**, **Int**, and **Bool**, indicating field elements, integers, and booleans respectively. As standard, compound types include array types $[T]$, denoting an array of elements of type T , and product types, $T_1 \times \dots \times T_n$, denoting tuples $(e_1, \dots, e_n)$

where each tuple element e_i has type T_i . Function types are of the form $x : \tau_1 \rightarrow \tau_2$, where τ_1 is the type of the input and τ_2 is the type of the output. The crucial part of the CODA type system are refinement types of the form $\{\nu : T \mid \phi\}$ which qualify standard types with an additional *logical qualifier* ϕ that the value must satisfy. As a simple example, the refinement type $\{\nu : \mathbb{F} \times \mathbb{F} \mid \nu.1 = \nu.2\}$ describes a pair whose first element is equal to its second element. In more detail, logical qualifiers in CODA are (possibly universally quantified) boolean combinations of atomic predicates involving terms t . Terms in the refinement type syntax include all expressions allowed in the CODA syntax as well as built-in functions like `toN` and `toZ` used for converting fields to signed and unsigned integers, respectively. We found such field-element-to-integer conversion functions to be crucial for concisely expressing many specifications of practical interest; hence, we include them as primitives in our type syntax. However, we also note that CODA's refinement term language is *extensible*, meaning that the developer can define additional functions to use as part of their specification.

Type Checking Rules

While refinement types allow developers to write specifications for their ZK circuits, we still need an algorithm that can be used to *statically verify* the type annotations, which is the job of the type checker. We formalize CODA's type checking algorithm using three kinds of typing judgments, namely (1) a circuit typing judgment, (2) an expression typing judgment, and (3) a subtyping judgment, which utilize the following environments:

- **Circuit store:** The *circuit store* Δ keeps track of previously defined circuits. In particular, it maps the name of a circuit to the input and output type and is used for type checking circuit references.

$$\begin{aligned}
 \text{isBaseType}(T) &= \begin{cases} \text{true} & \text{if } T \in \{\text{Int}, \text{F}, \text{Bool}\} \\ \text{true} & \text{if } T = [T'] \text{ and } \text{isBaseType}(T') \\ \text{true} & \text{if } T = T_n \times \dots \times T_n \text{ and } \text{isBaseType}(T_i) \text{ for } i \in \{1, \dots, n\} \\ \text{false} & \text{otherwise} \end{cases} \\
 \\
 \frac{\Delta; \Gamma \vdash e : \tau' \quad \Gamma \models \tau' <: \tau}{\Delta; \Gamma \vdash e : \tau} &\text{TE-SUB} \\
 \\
 \frac{}{\Delta; \Gamma \vdash \star : \{\nu : \text{F} \mid \text{true}\}} &\text{TE-NONDET} \qquad \frac{f \in \mathbb{F}_p}{\Delta; \Gamma \vdash f : \{\nu : \text{F} \mid \nu = f\}} \text{TE-CONSTF} \\
 \\
 \frac{n \in \mathbb{Z}}{\Delta; \Gamma \vdash n : \{\nu : \text{Int} \mid \nu = n\}} &\text{TE-CONSTINT} \qquad \frac{b \in \{\text{true}, \text{false}\}}{\Delta; \Gamma \vdash b : \{\nu : \text{Bool} \mid \nu = b\}} \text{TE-CONSTBOOL} \\
 \\
 \frac{\text{isBaseType}(T)}{\Delta; \Gamma \vdash \text{nil} : \{\nu : [T] \mid \text{length } \nu = 0\}} &\text{TE-NIL} \\
 \\
 \frac{\Gamma(x) = \{\nu : T \mid \phi\}}{\Delta; \Gamma \vdash x : \{\nu : T \mid \nu = x\}} &\text{TE-VAR} \qquad \frac{\Gamma(x) = \tau \quad \tau \equiv y : \tau_1 \rightarrow \tau_2}{\Delta; \Gamma \vdash x : \tau} \text{TE-VARFN} \\
 \\
 \frac{\Delta; \Gamma \vdash x : \text{F} \quad \Delta; \Gamma \vdash y : \text{F}}{\Delta; \Gamma \vdash \text{assert } x = y : \{\nu : \text{Unit} \mid x = y\}} &\text{TE-ASSERT} \\
 \\
 \frac{\Delta; \Gamma \vdash x : \text{Bool} \quad \Delta; \Gamma \vdash y : \text{Bool}}{\Delta; \Gamma \vdash x \odot y : \{\nu : \text{Bool} \mid \nu = x \odot y\}} &\text{TE-BINOPBOOL} \\
 \\
 \frac{\Delta; \Gamma \vdash x : \text{Int} \quad \Delta; \Gamma \vdash y : \text{Int}}{\Delta; \Gamma \vdash x \oplus y : \{\nu : \text{Int} \mid \nu = x \oplus y\}} &\text{TE-BINOPINT} \\
 \\
 \frac{\Delta; \Gamma \vdash x : \text{F} \quad \Delta; \Gamma \vdash y : \text{F}}{\Delta; \Gamma \vdash x \otimes y : \{\nu : \text{F} \mid \nu = x \otimes y\}} &\text{TE-BINOPFIELD} \\
 \\
 \frac{\Delta; \Gamma \vdash x : T \quad \Delta; \Gamma \vdash y : T}{\Delta; \Gamma \vdash x = y : \{\nu : \text{Bool} \mid \nu = (x = y)\}} &\text{TE-EQ}
 \end{aligned}$$

Figure 2.6: Selected typing rules for CODA expressions and programs

$$\begin{array}{c}
 \frac{\Delta; x : \tau_1, \Gamma \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1. e : (x : \tau_1 \rightarrow \tau_2)} \text{TE-ABS} \\
 \\
 \frac{\Delta; \Gamma \vdash x_1 : (s : \tau_s \rightarrow \tau_r) \quad \Delta; \Gamma \vdash x_2 : \tau_s}{\Delta; \Gamma \vdash x_1 \ x_2 : \tau_r[s \mapsto x_2]} \text{TE-APP} \\
 \\
 \frac{\Delta; \Gamma \vdash e_1 : T \quad \Delta; \Gamma \vdash e_2 : [T] \quad \Delta; \Gamma \vdash \{\nu : \mathbf{Unit} \mid \phi[\nu \mapsto e_1 :: e_2]\} : \{\nu : \mathbf{Unit} \mid \mathbf{true}\}}{\Delta; \Gamma \vdash e_1 :: e_2 : \{\nu : [T] \mid \phi\}} \text{TE-CONS} \\
 \\
 \frac{\Delta(C) = \mathbf{circuit} \ (x_1 : T_1) \cdots (x_n : T_n) \rightarrow (y : T_y)}{\Delta; \Gamma \vdash \#C : (x_1 : T_1 \rightarrow \cdots \rightarrow x_n : T_n \rightarrow T_y)} \text{TE-CREF} \\
 \\
 \frac{s, e, j, f, a \text{ fresh} \quad \tau_f = j : \{\nu : \mathbf{Int} \mid s \leq \nu < e\} \rightarrow \tau[i \mapsto j] \rightarrow \tau[i \mapsto j + 1] \quad \tau_a = \tau[i \mapsto s] \quad \tau_r = \tau[i \mapsto e]}{\Delta; \Gamma \vdash \mathbf{iter}_{\lambda i. \tau} : s : \mathbf{Int} \rightarrow e : \mathbf{Int} \rightarrow f : \tau_f \rightarrow a : \tau_a \rightarrow \tau_r} \text{TE-ITER} \\
 \\
 \frac{\Delta; \Gamma \vdash e : \{\nu : [T] \mid \mathbf{length} \ \nu \geq k\} \quad \Delta; \Gamma \vdash e_i : \{\nu : \mathbf{Int} \mid 0 \leq \nu < k\}}{\Delta; \Gamma \vdash e[e_i] : T} \text{TE-INDEX} \\
 \\
 \frac{\Delta; \Gamma \vdash e_i : \tau_i \quad \tau = \{\nu : \mathbf{Unit} \mid \phi[\nu.1 \mapsto x_1] \cdots [\nu.n \mapsto x_n]\} \quad x_i \text{ fresh for all } i \in \{1, \dots, n\} \quad x_n : \tau_n, \dots, x_1 : \tau_1, \Gamma \models \{\nu : \mathbf{Unit} \mid \mathbf{true}\} <: \tau}{\Delta; \Gamma \vdash (e_1, \dots, e_n)_\phi : \{\nu : \tau_1 \times \dots \times \tau_n \mid \phi\}} \text{TE-PRODCONS} \\
 \\
 \frac{u \text{ fresh} \quad \Delta; \Gamma \vdash e_1 : \{\nu : \tau_1 \times \dots \times \tau_n \mid \phi\} \quad \tau_u = \{\nu : \mathbf{Unit} \mid \phi[\nu.1 \mapsto x_1] \cdots [\nu.n \mapsto x_n]\} \quad \Delta; u : \tau_u, x_n : \tau_n, \dots, x_1 : \tau_1, \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash \mathbf{match} \ e_1 \ \mathbf{with} \ (x_1, \dots, x_n) \rightarrow e_2 : \tau} \text{TE-PRODDISTR} \\
 \\
 \frac{\Delta; x_1 : T_1, \dots, x_n : T_n \vdash e : T_y}{\vdash_C \mathbf{circuit} \ C \ (x_1 : T_1) \cdots (x_n : T_n) \rightarrow T_y \ \{e\}} \text{T-CIRCDDECL}
 \end{array}$$

Figure 2.6: Selected typing rules for CODA expressions and programs (continued)

- **Type environment:** The *type environment* Γ maps variable names to their types and is used to type check variable references.

Figure 2.6 presents our system’s typing rules. We elided some of the standard, uninteresting rules for space.

Circuit typing. The top-level typing judgment, labeled T-CIRCDECL, is used to type check CODA circuits. If the body e of the circuit has type T_y under the assumption that each argument x_i has type T_i , then the whole circuit is well-typed.

Expression typing. We describe well-typed expressions with the typing judgment $\Delta; \Gamma \vdash e : \tau$, meaning that “under context Γ , the expression e has type τ .” Due to space constraints, we only describe the most important expression typing rules in the following discussion; however, the interested reader can find all expression typing rules (together with their soundness proof) in Appendix A.2. We start with the standard rules first and work our way down to the more involved type checking rules.

Variables. The rules TE-VAR and TE-VAR-FUNC are used for typing variables. In both of these rules, the type of the variable is found in the context. For non-functional variables, we additionally retain the information that the returned data is the same as what is present in the variable itself. This enables simple, lightweight equality reasoning.

Constants. The rule TE-NONDET and TE-CONSTLIST are used for typing constants. Note that $\star$ corresponds to non-deterministically chosen field elements; hence, $\star$ always has type F . Furthermore, a field constant f has type $\{\nu : F \mid \nu = f\}$, which essentially corresponds to the singleton set $\{f\}$. (The typing rules for boolean and integer constants are similar.)

Assertions. The type checking rule for assertions, labeled TE-ASSERT, keeps track of the constraint being asserted in the type environment. Assertions only ensure the asserted information so the base type is **Unit**.

Binops. The rule TE-BINOPFIELD deals with binary expressions. For a given expression $x \otimes y$ to type check, x and y must both be field elements. The type of the expression is then $\{\nu : T \mid \nu = x \otimes y\}$. The elided binop rules, defined on data types like booleans, follow a similar structure.

Functions. The rules for function abstraction (TE-ABS) and application (TE-APP) are standard: The abstraction rule binds the type of x to its declared type τ_1 and checks that e_2 has type τ_2 under this assumption. The application rule $x_1 x_2$ checks that x_1 has function type $s : \tau_s \rightarrow \tau_r$ and that x_2 has type τ_s . Then, the resulting type of the application is the return type τ_r , with occurrences of s in τ_r substituted by x_2 .

Iteration. The type checking rule for the `iter` construct essentially checks the validity of the annotated loop invariant. In particular, recall that the `iter` construct is annotated with an invariant of the form $\lambda i. \tau$, indicating that the result of the aggregation has type τ for each index i . Hence, this rule asserts that (1) the initial value of the accumulator has type $\tau[i \mapsto s]$ (where s is the start index), (2) the return value has type $\tau[i \mapsto e]$ (where e is the end index), and (3) the function f has a suitable type of the form $\tau_f \equiv j : \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$, where τ_1 states that the index j is in the range $[s, e)$ and the return type $\tau_2 \rightarrow \tau_3$ states that the accumulator invariant τ is preserved, since $\tau_2 \equiv \tau[i \mapsto j]$ and $\tau_3 \equiv \tau[i \mapsto j + 1]$.

Circuit references. Circuit references are type-checked similarly to function applications. In particular, if C is a circuit with n inputs of type $T_1, \dots, T_n$ and an output of type τ_y , then the expression $\#C$ has type $x_1 : T_1 \rightarrow \dots \rightarrow x_n : T_n \rightarrow T_y$.

Subtyping

The type checking rules described in the previous section make use of a *subtyping* relation: Specifically, the TE-SUB rule from Figure 2.6 states that an expression of type τ' also has type τ as long as τ' is a subtype of τ , denoted as $\Gamma \models \tau' <: \tau$. In this subsection, we discuss the subtyping relation, presented in Figure 2.7, underlying CODA's

$$\begin{array}{c}
 \frac{}{\Gamma \models \tau <: \tau} \text{TSUB-REFL} \qquad \frac{\Gamma \models \tau_1 <: \tau_2 \quad \Gamma \models \tau_2 <: \tau_3}{\Gamma \models \tau_1 <: \tau_3} \text{TSUB-TRANS} \\
 \\
 \frac{P \equiv \llbracket \phi_1 \rrbracket \implies \llbracket \phi_2 \rrbracket \quad \Gamma \models T_1 <: T_2 \quad \forall \vec{x} \in \text{dom}(\Gamma). \forall \nu. \text{Encode}(\Gamma) \implies P}{\Gamma \models \{\nu : T_1 \mid \phi_1\} <: \{\nu : T_2 \mid \phi_2\}} \text{TSUB-REFINE} \\
 \\
 \frac{\Gamma \models \tau_y <: \tau_x \quad z \text{ fresh} \quad \Gamma \models \tau_r[x \mapsto z] <: \tau_s[y \mapsto z]}{\Gamma \models (x : \tau_x \rightarrow \tau_r) <: (y : \tau_y \rightarrow \tau_s)} \text{TSUB-FUN} \\
 \\
 \frac{\Gamma \models T_1 <: T_2}{\Gamma \models [T_1] <: [T_2]} \text{TSUB-ARRAY} \\
 \\
 \frac{\Gamma \models T_i <: T'_i \text{ for all } i \in \{0, \dots, n\}}{\Gamma \models T_1 \times \dots \times T_n <: T'_1 \times \dots \times T'_n} \text{TSUB-PRODUCT}
 \end{array}$$

Figure 2.7: Subtyping rules for CODA

type system.

The first two rules, TSUB-REFL and TSUB-TRANS, ensure that the subtyping relation is reflexive and transitive.

The third rule, called TSUB-REFINE, is the crux of the entire type system and reduces refinement type checking to querying logical validity in some formal logic. In particular, a refinement type $\{\nu : T_1 \mid \phi_1\}$ is considered a sub-type of $\{\nu : T_2 \mid \phi_2\}$ if the following two conditions hold: First, the base type T_1 must be a subtype of T_2 . Second, the logical qualifier ϕ_2 must be logically entailed by ϕ_1 under the type environment Γ . To perform the second check, the CODA type system generates a logical encoding of ϕ_1, ϕ_2 and Γ in Coq and produces a lemma that must be proven in order for type checking to succeed. We discuss the process of discharging proof obligations in Coq in the next subsection.

The next rule called TSUB-FUN is used for deciding sub-typing between functions. As

standard, this rule states that subtyping for functions is contravariant with respect to the argument types and covariant with respect to the return types. The next two rules for arrays and products are also standard and state that subtyping is covariant with respect to the nested element types.

2.3.4 Discharging Proof Obligations in Coq

Because logical qualifiers in the CODA type system make number-theoretic statements about finite field elements, the lemmas produced by the CODA type checker cannot be discharged automatically in general. Hence, we adopt a *semi-automated* approach, based on Coq, for discharging the proof obligations generated by the CODA type checker. We choose to use Coq for this purpose because of the existence of libraries for finite field arithmetic as well as a rich dictionary of tactics for proof automation.

Recall from the previous subsection that subtyping checks in the CODA type system may involve an application of **TSUB-REFINE** rule, which has the following premise:

$$\forall \vec{x} \in \text{dom}(\Gamma). \forall \nu. \text{Encode}(\Gamma) \implies \llbracket \phi_1 \rrbracket \implies \llbracket \phi_2 \rrbracket$$

This logical validity query is translated into a Coq proposition by translating the formulas ϕ_1, ϕ_2 as well as the typing environment into Coq objects. Hence, each application of the **TSUB-REFINE** rule during type checking corresponds to a Coq proposition that must be discharged, so type checking is only valid under the assumption that the resulting proof obligations have been discharged. As an example, the following is one of the proof obligations generated for the **BigLessThan** example from Section 2.2. This Coq proposition essentially states that the output signal, *out*, of the circuit actually indicates whether $|xs| < |ys|$, under the assumption that the **iter** invariant holds.

$$\begin{aligned}
 & \forall (k : \text{nat})(xs \ ys : \text{list } F)(out \ lt : F), \\
 & 2 \leq k \wedge \text{length}(xs) = \text{length}(ys) = k \\
 & \wedge \text{binary}(xs) \wedge \text{binary}(ys) \\
 & \wedge lt = |xs[:k]| <? |ys[:k]| \wedge out = lt \\
 & \implies out = |xs| <? |ys|.
 \end{aligned}$$

Discharging a proof obligation requires providing a formal Coq proof of the corresponding proposition. As stated earlier, Coq has a number of proof tactics that can be used to automate proofs, so some of the simpler proof obligations can be discharged automatically or with very little manual effort, particularly with the help of the Fiat-Crypto library [57]. For instance, 10 proof obligations for the **BigLessThan** example can be discharged in a fully automated way via Coq’s standard **auto** tactic, and an additional 3 may be discharged trivially using fewer than four standard tactics.

In addition to the standard tactics provided by Coq and the Fiat-Crypto library [57] (which we use for representing finite fields), we identified a number of other opportunities for automatically discharging common proof obligations generated by the CODA type checker. Table 2.1 summarizes some of the tactics that we found to be useful when proving CODA-generated Coq propositions. For example, because statements about integers operations must be implemented using field operations, which can overflow, we provide tactics that can automatically prove the field operations are overflow-safe. First, the **F__to__Z** tactic rewrites a field expression into an equivalent integer expression, generating integer inequality side-conditions that ensure overflow safety. However, the generated side-conditions often involve non-linear arithmetic, such as multiplication and

Table 2.1: Selected CODA Coq tactics.

Tactic	Summary
F_to_Z	Rewrite a field expression into an integer expression, generating overflow-safety side-conditions
overflow	Solve overflow-safety obligations
split_sum	Split a summation into disjoint sub-summations
switch_sum	Exchanges two nested summations
reduce_sum	Reduce trivial summations
ind	Perform case analysis on an binary field element

exponentiation, making automated solving undecidable. To mitigate this problem, we provide a domain-specific **overflow** tactic that performs backtracking search and attempts to reduce non-linear inequalities into equivalent linear inequalities based the user-provided bounds on existing variables. Finally, the resulting linear inequalities are solved by existing decision procedures such as Coq’s **lia** tactic. As another example, because ZKP programs build functionalities using field *arithmetics* such as addition and multiplication, the proofs often need to reason about indexed summations. To this end, we provide the **split_sum** tactic that splits a summation into multiple disjoint summations, **switch_sum** that exchanges two nested summations, and **reduce_sum** to reduce trivial summations into simpler expressions.

2.3.5 Properties of Well-Typed Coda Programs

In this section, we present the key correctness guarantees established by well-typed CODA programs and refer the reader to Appendix A.2 for the proofs. The correctness of CODA’s type system is characterized by expression type preservation. The preservation theorem states that if e is a well-typed expression with type τ and e evaluates to some value v , then v also has the type τ .

Type preservation requires one additional assumption – well-typed valuations. We describe well-typed valuations with the judgment $\Delta; \Gamma \models \sigma$, meaning that “under context

Γ , the values of all variables in σ have the same types as those in Γ ." The interested reader can find the formal definition of valuation typing in the Appendix A.2. This assumption ensures that the arguments provided to a circuit are well-typed, for if the arguments to the circuit are not well-typed then the resulting value may also not be well-typed.

Theorem 2.1 (Expression Type Preservation). *If $\Delta; \Gamma \vdash e : \tau$ and $\sigma \vdash e \Downarrow v$ and $\Delta; \Gamma \models \sigma$, then $\Delta; \Gamma \vdash v : \tau$.*

As a corollary, Theorem 2.1 immediately implies the following: If (1) C is a well-typed CODA program and $\vec{e}$ are input values satisfying the assumptions specified in the input types of C , and (2) $\#C \vec{e}$ evaluates to output values $\vec{v}$, then (3) the output values $\vec{v}$ are guaranteed to satisfy the output types of C , as stated in the following theorem:

Theorem 2.2 (Circuit Evaluation Type Preservation). *If $\Delta; \Gamma \vdash \#C \vec{e} : \tau$ and $\sigma \vdash \#C \vec{e} \Downarrow v$ and $\Gamma \models \sigma$, then $\Delta; \Gamma \vdash v : \tau$.*

$u ::=$ c $(u_1, \dots, u_n)$ $u_1 :: u_2$ $\text{Closure}(\lambda x : \tau. e, \sigma)$ r $r \otimes u \mid u \otimes r$	Compilation values: - constants - product value - array value - closure - R1CS variable - Irreducible expressions
$\sigma ::=$ $\emptyset$ $\sigma[x \mapsto u]$	Valuation: - empty valuation - augmentation

$\frac{}{\sigma \vdash u \rightsquigarrow \langle \emptyset, u \rangle} \text{C-VALUE}$	$\frac{r \text{ fresh}}{\sigma \vdash \star \rightsquigarrow \langle \emptyset, r \rangle} \text{C-NONDET}$
$\frac{\sigma \vdash e_1 \rightsquigarrow \langle \Phi_1, u_1 \rangle \quad \sigma \vdash e_2 \rightsquigarrow \langle \Phi_2, u_2 \rangle}{\sigma \vdash \text{assert } e_1 = e_2 \rightsquigarrow \langle \Phi_1 \cup \Phi_2 \cup \{u_1 = u_2\}, \text{unit} \rangle} \text{C-ASSERT}$	
$\frac{\sigma \vdash e_1 \rightsquigarrow \langle \Phi_1, u_1 \rangle \quad \sigma \vdash e_2 \rightsquigarrow \langle \Phi_2, u_2 \rangle \quad u_1, u_2 \in \mathbb{F} \quad u_1 \otimes u_2 = u_3}{\sigma \vdash e_1 \otimes e_2 \rightsquigarrow \langle \Phi_1 \cup \Phi_2, u_3 \rangle} \text{C-RED}$	
$\frac{\sigma \vdash e_1 \rightsquigarrow \langle \Phi_1, u_1 \rangle \quad \sigma \vdash e_2 \rightsquigarrow \langle \Phi_2, u_2 \rangle \quad u_1 \notin \mathbb{F} \text{ or } u_2 \notin \mathbb{F}}{\sigma \vdash e_1 \otimes e_2 \rightsquigarrow \langle \Phi_1 \cup \Phi_2, u_1 \otimes u_2 \rangle} \text{C-IRRED}$	
$\frac{r_i \text{ fresh for all } i \in \{1, \dots, n+1\} \quad [x_1 \mapsto r_1] \cdots [x_n \mapsto r_n] \vdash e \rightsquigarrow \langle \Phi, u \rangle \quad \Phi' = \Phi \cup \{r_{n+1} = u\}}{\text{circuit } C (x_1 : \mathbb{F}) \cdots (x_n : \mathbb{F}) \rightarrow \mathbb{F} \{e\} \rightsquigarrow_C \Phi'} \text{C-CIRC}$	

Figure 2.8: Rules for compiling CODA to R1CS constraints

2.4 The CODA Compiler

In this section, we describe the CODA compiler for converting CODA programs to R1CS constraints. Because zero-knowledge proof systems (based on zkSNARKs [60]) generate the prover and verifier from a Rank 1 Constraint system, the CODA compiler

needs to generate R1CS in order to leverage existing frameworks for zkSNARK generation. More formally, a Rank-1 constraint system (V, Φ) consists of a set of variables V and a list Φ of second-degree polynomial constraints of the form $A * B + C = 0$, where A , B and C are field constants or variables in V . The goal of the CODA compiler is to translate CODA programs to a set of R1CS constraints of this form.

We describe our compilation technique in Figure 2.8 using judgments of the following shape:

$$\sigma \vdash e \rightsquigarrow \langle \Phi, u \rangle$$

indicating that, under (symbolic) valuation σ , expression e evaluates to a *compilation value* u , with a corresponding set Φ of R1CS constraints. A compilation value u is essentially a partially evaluated version of expression e , where the original expression is simplified as much as possible and brought into an irreducible form. Because many CODA expressions are not valid R1CS terms, the primary goal of partial evaluation is to facilitate the generation of valid R1CS constraints.

Figure 2.8 shows a representative subset of the compilation rules underlying the CODA compiler. The first rule called C-NONDET simply introduces a fresh R1CS variable r representing a non-deterministically chosen field element. The next rule called C-ASSERT adds a new constraint $u_1 = u_2$ where u_1 and u_2 are the results of partially evaluating expressions e_1 and e_2 respectively. The next two rules C-RED and C-IRRED compile binary field expressions of the form $e_1 \otimes e_2$. In the reducible case, both e_1 and e_2 are partially evaluated as constants $(u_1, u_2 \in \mathbb{F})$, so we further reduce $e_1 \otimes e_2$ by evaluating $u_1 \otimes u_2$. Otherwise, in the irreducible case, the compilation result is the symbolic expression $u_1 \otimes u_2$.²

The final rule called C-CIRC in Figure 2.8 presents the compilation procedure for the

²Fresh intermediate variables are generated to reduce the degree of irreducible field expressions to ≤ 2 .

entire circuit. Given a circuit C with parameters $x_1, \dots, x_n$ and body e , the compilation procedure introduces fresh symbolic values r_i for each parameter x_i and then compiles e into (Φ, u) . The final R1CS encoding of the circuit is then obtained as $\Phi \cup r_{n+1} = u$, where the fresh variable r_{n+1} corresponds to the circuit output.

Due to lack of space, Figure 2.8 does not show the compilation rules for *all* CODA expressions, and the remaining compilation rules are basically the same as the operational semantics. Instead of formally describing the remaining compilation rules in detail, we remark on a few salient features of the remaining rules. First, because circuit inputs and the $\star$ expression in CODA correspond to field elements, all boolean and integer expressions can be compiled down to constants. As a corollary, CODA can unroll all loops during compilation. Furthermore, because array lengths in CODA are fixed, the CODA compiler can statically infer the exact shape of arrays. Hence, array operations, such as indexing and mapping, can also be fully evaluated.

2.5 Implementation

In this section, we provide more details about the implementation of CODA, which is largely implemented in OCaml and consists of around 4000 lines of source code.

The CODA language We implemented CODA as an embedded domain-specific language (EDSL) in OCaml. That is, CODA programs are expressed as OCaml expressions built from a library of atomic building blocks that represent CODA constructs. The choice of implementing CODA as an EDSL means that the developer is allowed to use features from the host language (e.g., OCaml’s type system, module system, as well as the expansive standard and third-party libraries) to significantly bootstrap the development speed.

Type Checking The type checker implements the type checking and subtyping rules shown in Figures 2.6 and 2.7. Before type checking, CODA programs are normalized to

the administrative normal form (ANF), and types are normalized to disallow consecutive refinements (i.e. refinement types of the form $\{\nu : \{\nu : T \mid \phi_1\} \mid \phi_2\}$ are normalized to $\{\nu : T \mid \phi_1 \wedge \phi_2\}$).

Lemma generation Recall that the `TSUB-REFINE` subtyping rule proves that one refinement type is the subtype of another by requiring that implication of their qualifiers is a valid Coq proposition. Thus, the CODA type checker collects all such implications generated during type checking, and translates them into equivalent Coq lemmas. The translation is mostly standard: a base type is embedded as the corresponding Coq type, an array type is embedded into Coq’s `list` type refined with its length, a product type is embedded as Coq’s product type, and a refinement type $\{\nu : T \mid \phi\}$ is embedded as T ’s embedding with a hypothesis that asserts the truth of the embedding of ϕ . We also define interpretations of CODA operators that may appear in ϕ . Finally, we use Fiat-Crypto [57] for the formalization of finite fields.

Extensibility of CODA Type System In case the user has difficulty encoding a desired property P using CODA’s refinement type (e.g., if P involves unbounded quantification), we allow the user to extend the type system by treating P as an opaque, uninterpreted predicate. The CODA type checker will run as normal, but it may generate proof obligations containing P . Thus, the user must also provide the interpretation of P in the form of a Coq definition.

Code generation The CODA compiler implements rules shown in Figure 2.8 to partially evaluate CODA programs into constraints over field expressions. We highlight some details that are elided in the compilation rules. First, the compiler relaxes the restriction on the circuit’s input type by allowing non-field inputs, as long as their concrete valuation has been provided. During compilation, the compiler resolves any array length into a concrete value such that array operations can be fully evaluated. Finally, the collected constraints undergo a global transformation pass that reduces the degree of each constraint to at

most 2, and eliminates any redundant constraints. Finally, the CODA compiler outputs an R1CS file in the zkInterface binary format for interoperating with different zk-SNARK backends.

Verified library Thanks to the functional aspect of CODA, the core CODA constructs can be easily composed to write expressive, higher-order *library* functions that can be reused across applications. Those functions can be verified just like circuits by checking the implementation against the specified types and proving the resulting lemmas in Coq. We have implemented this idea by building a verified library that encapsulates common ZKP patterns such as branching, functional zip, element-wise array arithmetics and array aggregate operations. Applications that use those library functions not only benefit from shorter, easier to understand programs, but also experience less proof burden, as those library functions have been fully certified.

2.6 Evaluation

In this section, we describe the results of an evaluation that is designed to answer the following research questions.

- **RQ1:** Can we use CODA to build formally verified implementations of commonly-used ZK circuits?
- **RQ2:** Can CODA help discover correctness bugs?
- **RQ3:** Can CODA proof automation tactics help reduce the manual verification effort?
- **RQ4:** Can the CODA compiler generate efficient R1CS code and how does it compare against Circom?

Benchmarks. To answer these questions, we collected 77 ZK circuits sampled from 9 widely-used libraries and projects implemented in Circom. The libraries include

`circomlib`, the standard Circom library, `circom-bigint`, a popular big-integer library for Circom, and the top-7 GitHub repositories under the “circom” topic ranked by the number of GitHub stars.

For each library, we selected the circuits used in our evaluation based on the following criteria: 1) **Documentation:** The original Circom code should have sufficient documentation to allow us to formally specify its behavior. 2) **External dependencies:** If the program has an external dependency, we require the external dependency to also satisfy the first criterion.

Experiment Setup. To perform our experiments, we manually translated these benchmarks to CODA in a way that preserves their semantics. We expressed the functional correctness requirements for each circuit as refinement type annotations and added invariant annotations wherever required in the CODA program. All experiments reported in this section are conducted on a MacBook Pro with Apple M1 Pro CPU and 16 GB of memory. CODA uses OCaml version 4.13.1 and Coq version 8.15.2. We use Circom 2.1.2 to compare with our generated R1CS constraints.

2.6.1 Main Results

Table 2.2 summarizes the program-size results of certifying our benchmarks using CODA. The column called “Original LOC” shows the lines of code of the original Circom circuit. In the “CODA Program” group, the columns “LOC”, “Spec” and “Spec/LOC” show the size of the translated program in terms of lines of code, the lines of specification (including the refinement type and loop invariants, if any), and the ratio of specification size to program size. Table 2.3 summarizes the proof-effort results. The “Proof Obligations” group displays statistics about the proof obligations generated by the CODA type checker: columns “Avg”, “Total” and “Provable” show the average number of proof obligations

Table 2.2: Program-size results: original Circom size, and the corresponding CODA program and specification size.

Library	Original LOC	CODA Program		
		LOC	Spec	Spec/LOC
Circomlib	114	37	21	0.57
Circom-bigint	264	74	38	0.51
Semaphore	71	23	22	0.96
Sismo	121	60	35	0.58
ZK-SBT	88	29	18	0.62
Darkforest-eth	29	10	5	0.50
ZK-SQL	47	12	4	0.33
Circomlib-ml	27	13	4	0.31
ed25519-circom	50	27	16	0.59

Table 2.3: Proof-effort results: per-library statistics on proof obligations generated by the CODA type checker, average Coq proof length, and type-checking time.

Library	Proof Obligations					Proof Length (avg lines)	TC Time (avg ms)
	Avg	Total	Provable	Discharged	Auto		
Circomlib	3	42	42	100%	76%	6	177
Circom-bigint	9	123	123	81%	69%	33	182
Semaphore	4	22	22	100%	91%	6	181
Sismo	8	32	32	100%	94%	3	184
ZK-SBT	6	37	37	100%	76%	4	181
Darkforest-eth	4	13	13	100%	77%	5	180
ZK-SQL	3	10	10	70%	80%	4	180
Circomlib-ml	3	10	10	83%	70%	4	181
ed25519-circom	3	17	9	100%	67%	6	182

per circuit, the total number of obligations, and the number of provable obligations respectively. The column called “Discharged” shows the percentage of correct lemmas that can be discharged (either automatically or interactively), and “Auto” shows the percentage of those that can be discharged fully automatically. Finally, the “Proof Length” column shows the average number of lines for the Coq proof, and the “TC Time” column shows the average running time of the type checker in milliseconds.

Our key result is that, for 93% of the Circom programs that are originally correct, we were able to prove the correctness of our CODA translation. Overall, the specification-to-implementation ratio for CODA program is 0.6, suggesting that CODA does not impose an undue burden on programmers for writing specifications. In terms of the burden of proving these specifications, the CODA type checker outputs an average of 5 proof obligations as Coq lemmas, of which 76% discharged fully automatically. For the remaining ones, the average proof length (using our tactic library) is 10 lines of Coq code.

Result for RQ1: CODA is able to certify 93% of the non-buggy benchmarks, achieving a program-to-specification ratio of 0.6 and requiring only 10 lines of manual proofs.

2.6.2 Vulnerability Discovery

Table 2.4: Vulnerabilities discovered in the benchmarks.

Library	Program	Bug
Circom-bigint	BigMod	Missing range check
Circomlib-ml	IsPositive	Zero considered positive
ed25519-circom	LessThanPower	Missing logical constraints
ed25519-circom	LessThanBounded	Missing logical constraints
ed25519-circom	fulladder	Missing logical constraints
ed25519-circom	onlycarry	Missing logical constraints

Because CODA is sound, a buggy program will lead to erroneous lemmas that will not be provable in Coq. When performing this evaluation, we discovered 6 benchmarks that resulted in proof obligations we were unable to discharge. Upon further inspection, we discovered that these failed proofs were due to subtle (and previously unknown) correctness bugs in the original Circom circuits.

Table 2.4 summarizes the vulnerabilities uncovered during our evaluation. In this table, the “Library” and “Program” columns show the library and program in which the

vulnerability resides, respectively. The “Bug” column summarizes the root cause of the vulnerability. Note that all of the uncovered bugs are due to important logical constraints that are missing in the original program. Hence, these vulnerabilities can be exploited by a malicious prover to produce bogus zero-knowledge proofs for untrue statements. We have informed the developers of all of the affected libraries of the vulnerabilities, and all issues have been confirmed by the developers.

Result for RQ2: CODA helped reveal 6 previously unknown vulnerabilities in widely-used ZK circuits.

2.6.3 Case Study: Reduction in Verification Effort

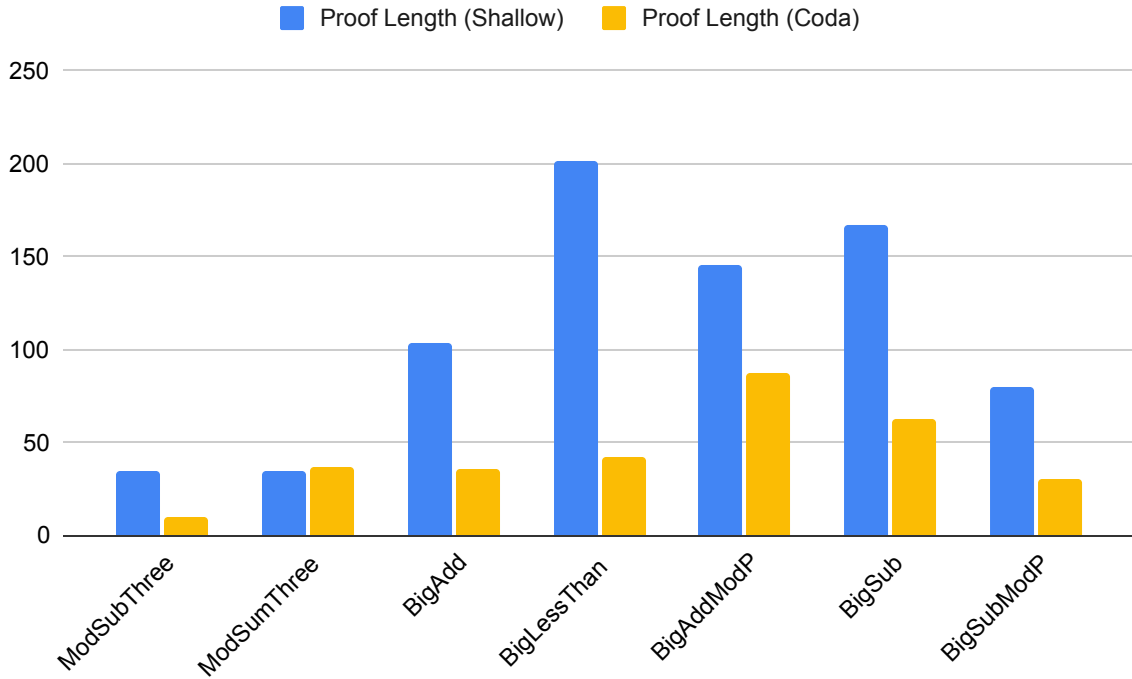


Figure 2.9: Comparison of verification effort of certifying the Circom-bigint library using CODA vs. using shallow embedding. The vertical axis shows the average number of constraints in log scale.

In this section, we investigate CODA’s effectiveness in reducing the verification effort.

Specifically, we compare the effort of certifying ZKP programs using CODA versus directly certifying the original Circom programs in Coq. In the latter approach, the source programs are syntactically translated into functions in Coq, a technique known as *shallow embedding*. For example, using shallow embedding, a Circom variable is encoded into a Coq variable, and the body of a circuit is encoded as a conjunction of Coq propositions.

We use the benchmarks from the Circom-bigint library as a case study. Figure 2.9 compares the verification effort required by each approach. The “Proof Length” columns show the number of lines of Coq proof that is required to certify the program using CODA and shallow embedding, respectively. On average, CODA enables 60% reduction in proof size.

Result for RQ3: CODA reduced the verification effort by requiring 60% shorter proofs compared to the shallow embedding approach.

2.6.4 Quality of Compiled R1CS

Finally, we investigate whether the CODA compiler is able to generate R1CS constraints whose size is comparable to the Circom compiler. Because the size of a constraint system crucially determines the cost producing the zero-knowledge proofs from the constraint system [59], a DSL that does not produce competitive R1CS constraints is unlikely to be adopted. To this end, we compare the number of constraints produced by the CODA compiler vs the Circom compiler. We exclude benchmarks that have transitive dependencies outside the benchmark set, as the compilation procedure requires all referenced circuits be defined. Additionally, if a circuit contains a natural number parameter, we instantiate the circuit using the representative values for said parameter and take the average across all parameter values. We compare CODA’s generated constraints against Circom’s constraints produced by optimization levels **-O1** and **-O2**.

The results of this evaluation are summarized in Figure 2.10. The key takeaway is that

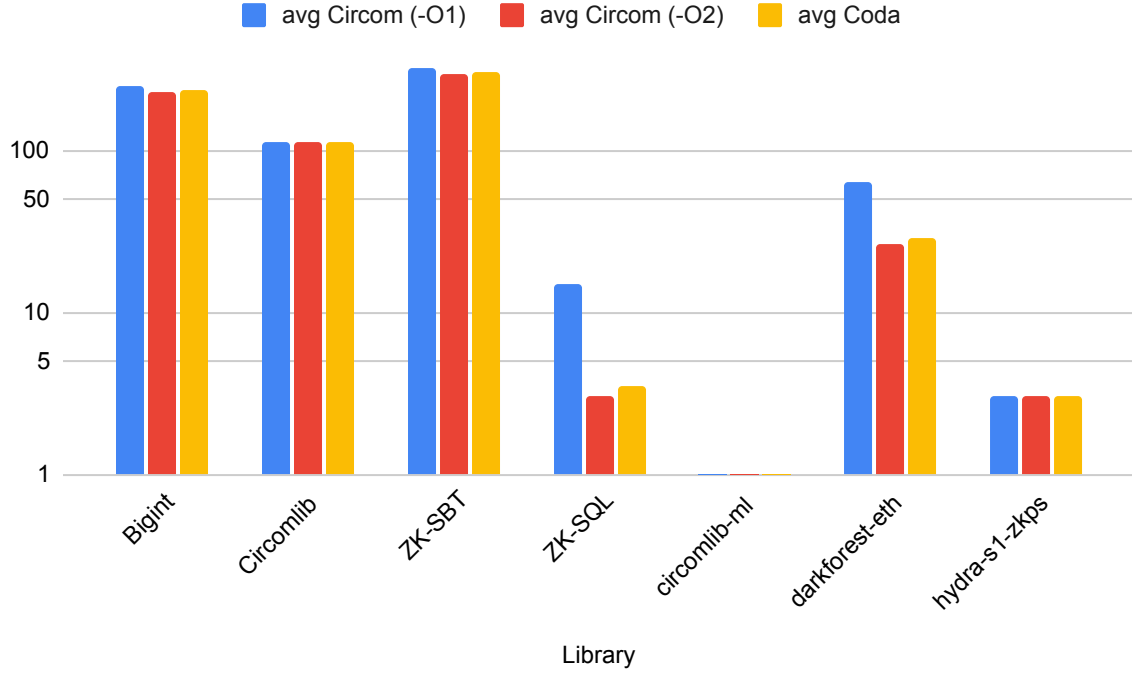


Figure 2.10: The quality of R1CS constraints generated by CODA compared to Circom.

the CODA compiler is able to produce R1CS constraints that are of comparable quality to the Circom compiler. In particular, compared to the Circom compiler on level **-O1**, CODA generates equal or fewer constraints for 100% of the benchmarks. Compared to the Circom compiler on level **-O2**, for 82% benchmarks, the number of CODA’s generated constraints is within 10% of the number of Circom’s constraints, while for 26% benchmarks, CODA produced equal or fewer constraints than the Circom compiler.

Result for RQ4: The CODA compiler can generate constraints whose quality is comparable to Circom.

2.7 Related Work

In this section, we survey prior work that is most closely related to our proposed approach.

Refinement types. There has been significant interest in using refinement types for formal verification [37, 38, 39, 40, 41]. Some example applications include proving the absence of integer overflows [41] and verification of memory safety [38]. Most of the prior techniques in this space keep type checking decidable and use an off-the-shelf SMT solver to discharge the constraints generated by the type checker. However, zero knowledge circuit verification is much more challenging than prior applications of refinement types, and existing SMT solvers do not provide adequate support for reasoning about polynomial equations over finite fields. Thus, in contrast to prior work, CODA outputs Coq propositions during type checking and uses a combination of automated tactics and manual intervention (where necessary) to enable formal verification of ZK circuits.

ZK programming languages and compilers. Due to the increasing demand for zero-knowledge proofs, there have been several new domain-specific languages (DSLs) targeting this domain. Example DSLs for constructing zkSNARKS [61] include Leo [25], Noir [36], Zokrates [26], Zinc [62], Snarky [63], Lurk [64], and CirC [27]. In contrast to these prior efforts, the main focus of CODA is on *verifiability* of the circuit with respect to the specification.

While most languages, include CODA, are intended for zkSNARKs, there are also other programming languages targeting zkSTARKs (Scalable Transparent ARgument of Knowledge) [14]. For instance, Cairo [15], is a Turing complete language that allows general computation, and has recently become quite popular for zkSTARKs.

Formal methods & ZK. Writing correct zero-knowledge programs requires specialized domain expertise. Due to the importance of ZK circuits in blockchain applications, recent work has proposed techniques for finding bugs in this domain [18, 19, 20, 21]. For instance, Circomspect [31] is an open-source static analyzer designed to find bugs in Circom programs. Circomspect performs syntactic pattern matching on the Circom source code to detect potential issues. However, this approach suffers from both false positives

as well as false negatives and is intended as a bug finder rather than a verifier. Wen et al. [33] study common vulnerabilities in Circom and describe a static analysis framework for detecting these vulnerabilities. Their technique operates over an abstraction called the *circuit dependence graph (CDG)* that captures key properties of the circuit and allows expressing *semantic vulnerability patterns* as queries over the CDG abstraction. Another recent effort [32] focuses on finding underconstrained bugs (or proving the absence thereof) in Circom programs. This approach combines SMT solving with lightweight uniqueness inference to effectively reason about underconstrained circuits. In contrast, Coda is designed with a refinement type system to specify the functional correctness of circuits. Unlike previous approaches that focus on finding common vulnerabilities, the focus of CODA is on verifying full correctness of the circuit with respect to a formal specification. Certification is achieved using a combination of type checking and semi-automated Coq proofs. On a related note, [4] formally verifies the equivalence between constraint systems for probabilistic proofs, with the goal of reducing the number of constraints.

Another orthogonal line of work [65] combines formal methods and the zero-knowledge domain in a different way. In particular, recent work has proposed ZK-UNSAT, a technique for proving that a propositional formula is unsatisfiable while revealing minimal information about the resolution refutation proof of unsatisfiability. As the focus of this work is generating zero-knowledge arguments of unsatisfiability, it is not directly related to CODA.

Finally, there have been works on formally verifying the *compilation* from high-level DSLs to arithmetic constraints. The Leo [25] programming language seeks to provide a certifying compiler from a high-level language to R1CS constraints. [34] develops a certified compiler based on CompCert [66] that correctly compiles a subset of C to quadratic constraints. These works are complementary to CODA in that they can be adapted to formally certify the CODA compiler.

Formal verification for cryptography. There is a body of work on applying formal verification techniques to cryptographic protocols. For instance, FiatCrypto [57] introduces a new approach for implementing cryptographic arithmetic in high-level code with machine-checked proofs of functional correctness through Coq. Corin et al. [67] leverage a variant of probabilistic Hoare logic to prove the security of ElGamal; Gagne et al. [68] use similar methods to prove the security of the front-end of many CBC-based MACs, PMAC and HMAC. Tiwari et.al [69] leverage component-based program synthesis to automatically generate padding-based encryption schemes, and block cipher modes of operations. EasyCrypt [70] is a toolset that allows users to specify and prove the correctness of cryptographic protocols.

On the other hand, there is little work on reasoning about the correctness of zero-knowledge proofs. Leveraging the MPC-in-the-head paradigm (i.e., Multi-Parity Computation), Sidorenco et al.[71] generated machine-checked proofs of ZK protocols using EasyCrypt. Almeida et al. [72] developed a certifying compiler for Σ -protocols, a broad class of zero-knowledge protocols including zkSNARKs [61]. Specifically, given a protocol written in its high-level language, the compiler generates an executable implementation that is provably correct using the Isabelle/HOL [73] theorem prover. More recent work has focused on building *specialized solvers* for polynomial equations over finite fields [42, 43, 44, 32]. Similar to the custom Coq tactic library that reduces the proof burden in CODA, we believe the above-mentioned works are complementary and can be incorporated into CODA to improve the degree of proof automation.

2.8 Conclusion

We presented CODA, a new statically-typed language for building zero-knowledge applications. CODA enables developers to *formally specify* and *statically check* properties

of ZK circuits through a rich refinement type system. The type checker underlying CODA generates a set of Coq lemmas that, if valid, collectively constitute a proof of correctness of the circuit. CODA facilitates interactive verification of these lemmas using a combination of domain-specific proof tactics and manual intervention where necessary. Finally, the CODA compiler automatically translates high-level CODA circuits to low-level R1CS constraints from which a zero-knowledge proof system can be generated using existing techniques.

We used CODA to formally specify and verify 77 existing circuits from widely-used Circom projects. We were able to verify the correct circuits and identified 6 previously unknown vulnerabilities in the original Circom circuits.

Chapter 3

Tessel: An Optimizing Compiler for Efficient Zero-Knowledge Circuits

3.1 Introduction

In this and the next chapter, we turn our attention to the compilation of ZK DSLs. Traditionally, there is a tradeoff between programmability/correctness and efficiency. Low-level circuit-based languages, such as Halo2 [23] and Circom [22], provide fine-grained control, enabling expert developers to optimize performance, but at the cost of steep learning curves and a higher risk of programming errors. Conversely, high-level domain-specific languages (DSLs) like Noir [24] and Leo [25] abstract away many complexities, making ZK circuit development more accessible—similar to mainstream languages like Rust [74]—but often at the expense of efficiency.

This tradeoff presents a fundamental challenge in ZK programming. Whereas CODA addresses *correctness* by giving programmers a refinement-type discipline to reason about the semantics of their circuits, we now look at what compilers for high-level DSLs can do after a program is written: how they can make the resulting low-level arithmetic

circuits *efficient*. When ZK programs are compiled, the resulting circuits often suffer from suboptimal performance in key metrics such as circuit size. Large or inefficient circuits not only increase computational overhead but also incur higher costs when deployed on blockchain platforms, where scalability and performance are critical concerns.

To address this challenge, this chapter proposes TESSEL, a compiler framework that enables the translation of high-level ZK programs into efficient low-level circuits, specifically targeting PLONKish arithmetization [10]. PLONKish circuits are widely used due to their flexibility in handling customized constraints, making them a popular choice for optimizing prover performance and reducing costs. The key insight of our approach is to treat this translation process as a multi-stage optimization problem. We begin by applying *partial evaluation* to simplify high-level language constructs (e.g., loops and arrays) while preserving optimization hints through a symbolic tracing algorithm. Instead of directly optimizing the low-level circuit, we introduce an intermediate representation (IR) called the *tabular representation*, which exposes the yet-to-be-determined *layout* of the circuit and enables a precise formulation of the *optimal arithmetization* problem, aiming to minimize circuit size. As we demonstrate in our evaluation, a direct encoding of this problem results in complex non-linear constraints that are difficult for off-the-shelf solvers to handle. As a first step in tackling this problem, we propose a simple *layout algorithm*, which improves efficiency while preserving the semantic integrity of the original ZK program.

We evaluate TESSEL on 26 real-world ZK programs. Our experiments demonstrate that TESSEL can generate arithmetic circuits that outperform mainstream compilers in some cases, and achieves a speedup of 1.81–4.44 $\times$. Through an ablation study, we show that TESSEL’s optimization techniques are essential for generating efficient circuits.

In summary, this chapter makes the following contributions:

- An optimizing compiler pipeline that transforms high-level ZK programs into efficient PLONKish arithmetic circuits.
- A formalization of the layout problem based on a tabular IR, and a heuristic layout algorithm that can solve the problem in practice.
- Empirical evaluation of our tool on a set of ZK programs, demonstrating improvements in circuit size and efficiency in comparison to existing compilers in certain cases.

3.2 Background and Overview of TESSEL

In this section, we briefly review some background on zero-knowledge proofs and PLONKish arithmetization.

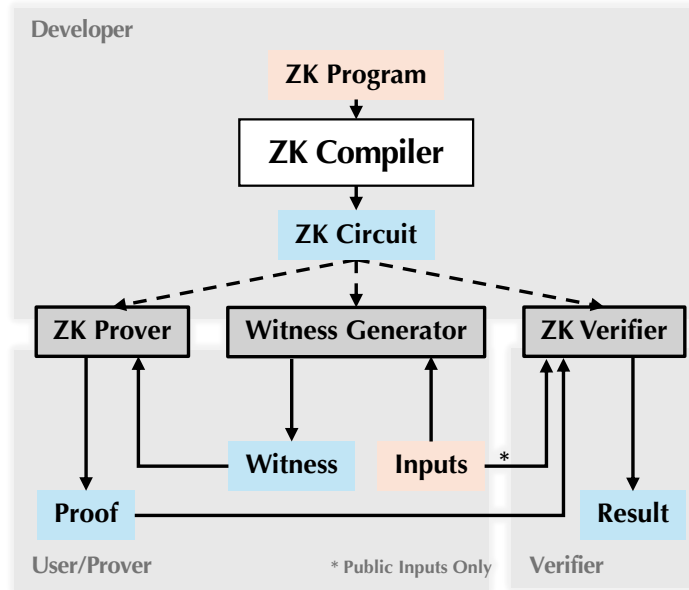


Figure 3.1: A typical compilation workflow of a ZK program.

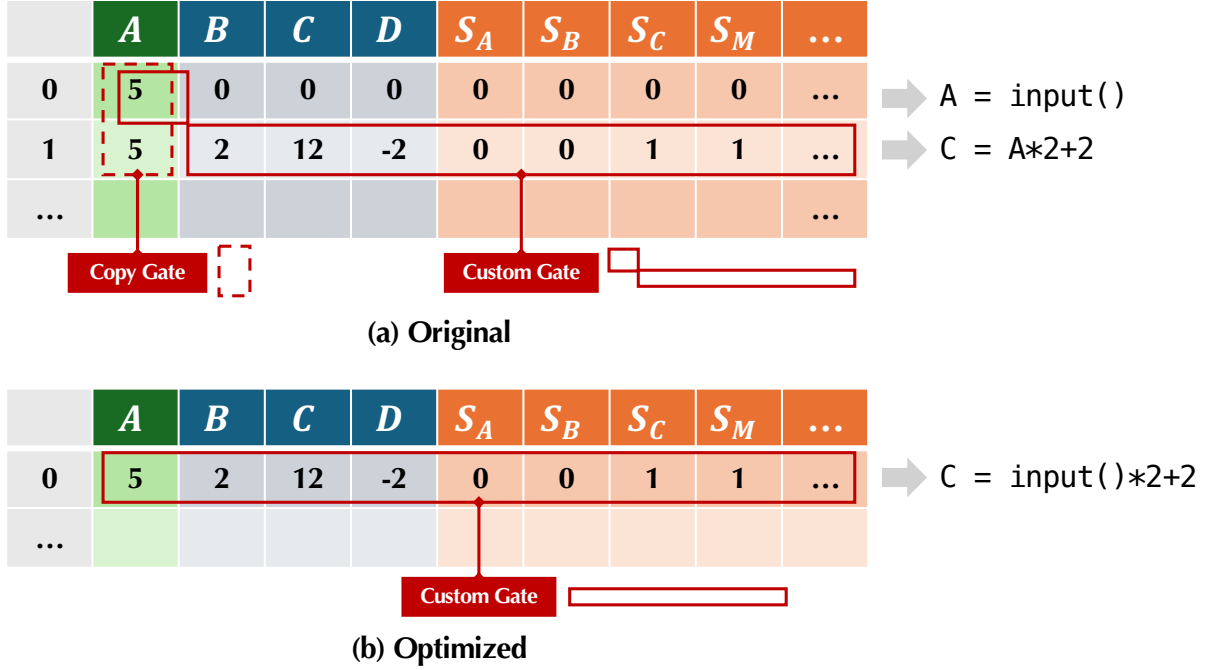


Figure 3.2: Two different schemes (original v.s. optimized) of PLONKish arithmetization of the same ZK program, with different types of columns and gates.

Zero-Knowledge Proofs Zero-knowledge proofs (ZKPs) are cryptographic protocols that enable one party to prove the validity of a statement to another without revealing any additional information. Various types of ZKPs, such as zk-SNARKs [61] and zk-STARKs [75], offer different trade-offs in efficiency and security, but the performance of any ZK application largely depends on the design of its compiler.

As shown in Figure 3.1, a typical ZK compilation workflow involves three key roles. First, the **developer** defines the statement to be proven and encodes it into a **ZK program** using specialized languages like Circom [22], Leo [25], and Halo2 [23]. This program is then compiled into a low-level arithmetic circuit, which is further transformed into three cryptographic components: a witness generator, a prover, and a verifier. Next, the **user**, also referred to as the prover, executes the ZK program with specific inputs using the witness generator, which produces an encoded representation of key values

known as the **witness**. The **prover** then uses this witness to generate a cryptographic proof that can be transmitted without exposing the underlying data. Finally, the **verifier** checks whether the proof is valid against a set of public inputs, confirming that the original statement holds without learning any private details.

At its core, a ZK program defines a computation that is transformed by the compiler into a structured ZK circuit—a set of polynomial equations that maintain the semantics of the original program. These circuits form the foundation for generating efficient proofs and play a crucial role in balancing security and performance in real-world applications.

PLONKish Arithmetization A cryptographic structure in a ZK circuit is usually expressed through a restricted set of operators (e.g., finite field operators), which could be syntactically different from its corresponding ZK program. We refer to such a process that connects a ZK program to its corresponding cryptographic structure as *arithmetization*. For example, ZK programs for PLONK-style proof systems (e.g., PLONK [10], Halo2 [23], UltraPLONK [13], etc.) share an inherent arithmetization in the form of polynomial constraints:

$$S_A(X) \cdot A(X) + S_B(X) \cdot B(X) + S_M(X) \cdot A(X) \cdot B(X) = S_C(X) \cdot C(X) + D(X),$$

where X is a table that stores inputs and intermediate values that participate in the computation of the statement to prove, and $A(\cdot)$, $B(\cdot)$, $C(\cdot)$, $D(\cdot)$ and $S_*(\cdot)$ retrieve cells from corresponding columns of a given table. Since arithmetization can be partially done by developers (e.g., Circom [22] and Halo2 [23]) or entirely tracked by ZK compilers (e.g., Noir [24], SP1 [29] and Jolt [30]), different schemes of arithmetization can result in different performance of a circuit, e.g., prover time, proof size and verifier cost.

Figure 3.2 shows an example table visualization of how PLONKish arithmetization

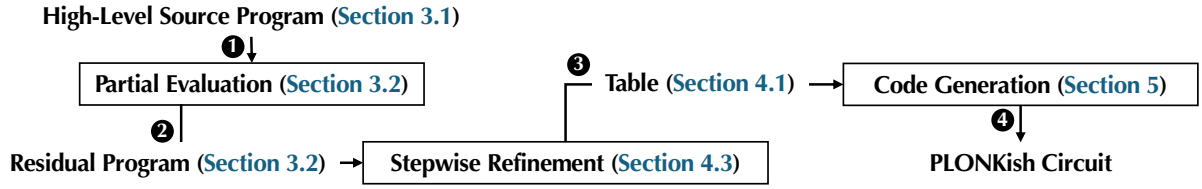


Figure 3.3: The compilation pipeline of TESSEL.

encodes computation expressed in ZK programs. In particular, as shown by the original table, cells from column A correspond to the inputs and should be filled by the user; cells from other columns are pre-assigned with particular values that encode the semantics of the ZK program. For example, the assignment of A in $A = \text{input}()$ doesn't utilize other cells in row 0, and the follow-up computation $C = A \cdot 2 + 2$ utilizes cells from all variable columns B , C and D , which is arithmetized as:

$$0 \cdot 5 + 0 \cdot 2 + 1 \cdot 5 \cdot 2 = 1 \cdot 12 - 2.$$

This constrains the value of C to be twice the value of A at the previous cell plus 2. Note that such computation utilizes the previous version of A , which should be retrieved from the last row. Since cells that are involved in this computation are distributed across different rows, a data structure called *gate* is defined to retrieve them. Similarly, since values of A do not change as traces are computed, a special copy gate is applied to enforce the equivalence of cells in column A .

A closer look at the original version (a) reveals inefficiencies, as the first row occupies multiple underutilized cells, unnecessarily increasing the finalized proof size. In the optimized version (b), the assignment of A is merged with the subsequent computation, eliminating an extra row and reducing overhead. In real-world circuits, where large numbers of columns and rows encode long computational traces, an optimal layout significantly improves proof size, proving efficiency, and verification speed. However, automatically

synthesizing such layouts remains a challenging task for existing ZK compilers.

From the above example, optimizing PLONKish arithmetization reduces to a table optimization problem, where rows encode execution steps and columns define variables, intermediates, and selectors. A well-optimized layout minimizes redundancy, reduces gate count, and improves prover efficiency. By structuring arithmetization this way, we compact constraints using partial evaluation and layout optimization, leading to more efficient circuits.

Overview of TESSEL Figure 3.3 shows an overview of the compilation pipeline for circuit optimization. The compiler takes source programs written in a typical high-level domain-specific language (DSL) for ZK programming, such as Noir [24]. The syntax and the semantics of the high-level language are explained in Section 3.3.1. The compiler first performs *symbolic tracing* to record domain-specific effects, while using *partial evaluation* to perform compile-time computation and eliminate high-level abstractions (Section 3.3.2). Then we transform the symbolic trace into an intermediate representation (IR) called *table IR*, which enables us to reduce the optimization of the circuit to the layout of the table, which we then address by a layout algorithm described in Section 3.4.3. Finally, we briefly discuss code generation into PLONKish arithmetic circuits in Section 3.5.

3.3 Preprocessing ZK Programs for Layout Optimization

3.3.1 High-Level Source Language

The syntax of the high-level source language is shown in Figure 3.4. This language distills popular high-level ZK DSLs (e.g. Noir [24]) into a core set of constructs. Notably,

$e ::=$	Expression:	$c ::=$	Statement:
$f \in \mathbb{F}$	field constant	$\text{let } x = rc$	let-binding
x	variable	$x := e$	assignment
$e \oplus e$	binary operation	$x[e] := e$	array write
$x[e]$	array read	$c; c$	sequencing
$r ::=$	Initializer:	$\text{repeat } e \{c\}$	loop
e	pure expression	$\text{if } e \text{ then } c \text{ else } c \text{ join}$	conditional
$\text{input}()$	input	$\text{constrain } e = e$	constraint
$[e; e]$	new array	$\oplus \in$	$\{+, -, \times\}$
$\star$	non-deterministic choice		

Figure 3.4: Syntax of the high-level source language.

besides standard imperative programming features such as mutable variables, arrays, and loops, the language includes primitive operations on finite fields, the non-deterministic choice operator $\star$, and the ability to constrain two finite field expressions to be equal using **constrain** $e = e$.

Semantically, a ZK program can be given a *relational* interpretation: each $\star$ is interpreted as non-deterministically choosing an arbitrary finite element that collectively satisfies the polynomial equality constraints imposed via the **constrain** $e = e$ statements. Equivalently, one can also interpret the program as building a *constraint system*, where each $\star$ introduces a fresh existential variable ranging over all elements in the underlying finite field, and each **constrain** $e = e$ statement accumulates a polynomial equation to some global constraint store.

Example 3.1. As our running example, Figure 3.5 shows a simple program that accepts two inputs x and z , and verifies that third partial sum in the harmonic series starting at $1/x$ is equal to z . That is, it checks whether $\sum_{i=x}^N \frac{1}{i} = z$. The program is mostly straightforward just like a typical imperative program, except for variable `inv`: because a constraint system can only contain polynomial – not rational – equations, divisions like $1/i$ are expressed indirectly. Specifically, on line 6, the program non-deterministically chooses a variable `inv` that is enforced to be the multiplicative inverse of i via the constraint `i * inv == 1` on line 7.

```

1 let x = input() in
2 let z = input() in
3 let sum = 0 in
4 let i = x in
5 repeat N {
6     let inv = ★ in
7     constrain i * inv == 1;
8     sum := sum + inv;
9     i := i + 1
10 };
11 constrain sum == z

```

Figure 3.5: An example ZK program in the high-level source language that checks whether the N -th partial sum in the harmonic series starting at $1/x$ is equal to input z .

```

1 write x ← input()
2 write z ← input()
3 write sum ← 0
4 write i ← read x
5 write inv ← ★
6 read i * read inv == 1
7 write sum ← read sum + read inv
8 write i ← read i + 1
9 write inv ← ★
10 read i * read inv == 1
11 write sum ← read sum + read inv
12 write i ← read i + 1
13 read sum == read z

```

Figure 3.6: The residual of the running example from Figure 3.5.

$v ::=$	Symbolic Value:	$t ::= (c)^*$	Trace
read $l \in \text{Loc}$	memory read	$c ::=$	Statement:
$f \in \mathbb{F}$	field constant	write $l \leftarrow r$	memory write
$v \oplus v$	binary operation	constrain $v = v$	constraint
		$r ::= v \mid \text{input}() \mid \star$	R-value

Figure 3.7: Syntax of residual programs.

$\llbracket x \rrbracket_{\nu, h} = f$	if $\nu(x) = l$ and $h(l) = f \in \mathbb{F}$
$\llbracket x \rrbracket_{\nu, h} = \text{read } l$	otherwise
$\llbracket e_1 \oplus e_2 \rrbracket_{\nu, h} = \llbracket \oplus \rrbracket(f_1, f_2)$	if $\llbracket e_1 \rrbracket_{\nu, h} = f_1$ and $\llbracket e_2 \rrbracket_{\nu, h} = f_2$
$\llbracket e_1 \oplus e_2 \rrbracket_{\nu, h} = \llbracket e_1 \rrbracket_{\nu, h} \oplus \llbracket e_2 \rrbracket_{\nu, h}$	otherwise
$\llbracket e \rrbracket_{\nu, h} = e$	if e is an R-value
$\llbracket x := e \rrbracket_{\nu, h} = (\text{write } l \leftarrow v, h[l \mapsto v])$	if $\nu(x) = l$ and $\llbracket e \rrbracket_{\nu, h} = v$
$\llbracket \text{constrain } e_1 = e_2 \rrbracket_{\nu, h} = (\cdot, h)$	if $\llbracket e_1 \rrbracket_{\nu, h} = \llbracket e_2 \rrbracket_{\nu, h} = f \in F$
$\llbracket \text{constrain } e_1 = e_2 \rrbracket_{\nu, h} = \perp$	if $\llbracket e_1 \rrbracket_{\nu, h} = f_1, \llbracket e_2 \rrbracket_{\nu, h} = f_2, f_1 \neq f_2$
$\llbracket \text{constrain } e_1 = e_2 \rrbracket_{\nu, h} = (\text{constrain } v_1 = v_2, h)$	if $\llbracket e_1 \rrbracket_{\nu, h} = v_1$ and $\llbracket e_2 \rrbracket_{\nu, h} = v_2$
$\llbracket c_1; c_2 \rrbracket_{\nu, h} = (s_1; s_2, h'')$	if $\llbracket c_1 \rrbracket_{\nu, h} = (s_1, h')$ and $\llbracket c_2 \rrbracket_{\nu, h'} = (s_2, h'')$
$\llbracket \text{repeat } x \{c\} \rrbracket_{\nu, h} = (\cdot, h)$	if $\llbracket x \rrbracket_{\nu, h} = 0$
$\llbracket \text{repeat } e \{c\} \rrbracket_{\nu, h} = (t; t', h'')$	if $\llbracket e \rrbracket_{\nu, h} = f \neq 0, \llbracket c \rrbracket_{\nu, h} = (t, h')$ and $\llbracket \text{repeat } (f - 1) \{c\} \rrbracket_{\nu, h'} = (t', h'')$

Figure 3.8: The partial evaluation algorithm.

3.3.2 Partial Evaluation

As the first step towards compiling the high-level source program into efficient PLONK-ish circuits, we leverage a custom partial evaluation algorithm [76] to record important domain-specific effects – namely, memory and constraint operations – that will later serve as important *hints* to the layout passes (Section 3.4.3). The algorithm also performs as much compile-time computation as possible, e.g., statically evaluating array operations and loop bounds, and propagating constant expressions. The result of this pass is a *residual program*, whose syntax is shown in Figure 3.7.

The core of the partial evaluation algorithm is presented in Figure 3.8. The algorithm assumes the following semantic domains:

- $v \in \text{Val}$ of symbolic values
- $l \in \text{Loc}$ representing mutable variables and array locations
- $a \in \text{Arr}$ of array values, which model arrays as functions $a : \mathbb{N} \rightarrow \text{Loc}$ from indices

to memory locations

- $\nu : \text{Var} \rightarrow \text{Val}$ that maps lexically scoped variables to values
- $h : \text{Loc} \rightarrow \text{Val}$ that maps memory locations to values.

The notation $\llbracket e \rrbracket_{\nu, h} = v$ denotes the partial evaluation of expression e under variable mapping ν and memory h , resulting in symbolic value v . The notation $\llbracket c \rrbracket_{\nu, h} = (t, h')$ denotes the execution of statement c under variable mapping ν and memory h , which updates the memory to h' while producing a residual program t that records important domain-specific effects of executing c .

The algorithm works as follows.

- If a variable x is mapped to some location l that holds a constant value, partial evaluation can read and return that constant value (the first rule). Otherwise, it returns a symbolic memory read expression `read  $l$` .
- Binary operations are evaluated if both operands are constants; otherwise, a symbolic expression is returned.
- The algorithm handles assignments and constraints by recording the corresponding memory write and constraint effects. Note that in a constraint statement `constrain  $e_1 = e_2$` , if both e_1 and e_2 evaluate to the same field constant f , then the constraint is satisfied at compile time and can be pruned. If they evaluate to different constants, then the program is unsatisfiable and no residual can be produced. Otherwise, the constraint is recorded in the residual program.
- The sequential composition of statements is handled by concatenating the residual programs produced by each statement, while loops are unrolled at compile-time.

Example 3.2. Figure 3.6 shows the result of running the partial evaluation algorithm on the example program from Figure 3.5 by instantiating the loop parameter N with 2.

The residual program is flattened into atomic steps, each recording a memory write or a constraint from execution.

3.4 Layout-Aware Circuit Optimization

Recall that optimization of a circuit’s arithmetization can be reduced to finding an optimal layout of its corresponding table representation with respect to certain cost metrics (e.g., table size). In this section, we address this problem first with a definition of the *layout problem* (Section 3.4.1), then with an algorithm for optimizing the layout given a residual program (Section 3.4.3).

3.4.1 The Layout Problem

A residual program eventually needs to be turned into a tabular representation of its arithmetization, a process we refer to as *layout*. Layout is crucial for the efficiency of the generated circuits, since the prover cost is directly affected by the size of the resulting table, as well as other factors determined by layout. Unfortunately, existing compilers for high-level ZK DSLs do not single out layout as an important problem that has wide-ranging performance implications, and instead rely on simple heuristics to generate circuits. A key contribution of this work is to formalize the layout problem and propose a layout algorithm to solve it. Specifically, given a residual program obtained by partially evaluating the source program, we approach the layout problem by minimizing certain cost metrics (e.g., table size) of the circuit’s arithmetization via our proposed *table IR*.

Layout The goal of layout is to turn a given circuit’s residual program into a table that assigns each cell to a specific row and column and synthesizes *gates* that refer to those

$g ::=$		Gate Expression:
	$c\langle i \rangle$	relative cell reference ($ i \leq m - 1$)
	$f \in \mathbb{F}$	field constant
	$g_1 \oplus g_2$	binary op

Figure 3.9: Syntax of gate expressions in table IR.

cells using offsets. Layout is a challenging problem due to the myriad of trade-offs involved and the complexity of the cost model underlying PLONKish proof systems [23, 77]. For example, the compiler needs to carefully balance the locality (the relative distance between cells referenced by a gate), copying (which improves locality at the expense of larger table size), expression complexity (the degree of the polynomial expression for a computation gate), and customization (the use of custom gates [10] to optimize the circuit for a specific program¹). Existing compilers [24, 79] sidestep the complexity by making significant sacrifices to keep the layout algorithm *simple*. In this work, however, we shall carefully formalize it as an instance of a constrained optimization problem.

We first precisely define the expected output of layout using the table IR, shown in Figure 3.9. The syntax of tables is parameterized by the set $\underline{m} = \{0, \dots, m - 1\}$ of row indices.² A table consists of a set of gate expressions $e \in E$, and a binary *selector* function $s : E \times \underline{m} \rightarrow \{0, 1\}$ that determines if a gate expression should be enabled at a specific row by returning 1 (enabled) or 0 (disabled).

A gate expression g is a polynomial over field $\mathbb{F}$ with variables that are *relative cell references*, written as $c\langle i \rangle$. The index i specifies the relative offset from the current row to the cell being referenced, and can be any integer modulo the size of the table. For example, the expression $c\langle -1 \rangle$ refers to the cell one row *above* the current cell in column c , while $c\langle 2 \rangle$ refers to the cell two rows *below*.

The layout problem can then be described as follows. Given a residual program t of

¹Similar to how just-in-time (JIT) compilers [78] can optimize for the specific program being executed.

²For simplicity, we assume tables have single columns, but our formalization can be extended to tables with multiple columns.

the target ZK circuit, we aim to find a natural number m representing the number of rows in the table, a set G of gate expressions, and a table $B : G \rightarrow \mathcal{P}(\underline{m})$ that maps each gate expression $g \in G$ to a set of rows R at which g is enabled.

We quantify the table size $\varsigma(B)$ as the cost metric of the optimization problem:

$$\varsigma(B) = |G_{\text{enabled}}| \cdot m,$$

where $G_{\text{enabled}} = \{g \in G \mid B(g) \neq \emptyset\}$ is the set of gate expressions that are enabled at least once in the table. Thus, to find an optimal layout for a circuit given its residual program t , we are essentially looking for a table B that minimizes the cost $\varsigma(B)$:

$$B^\diamond = \arg \min_B \varsigma(B) \tag{3.1}$$

while subject to the following hard constraints (specified by the developer):

- $d_B \leq d_{\max}$: The maximum degree of polynomials in B should be no larger than $d_{\max}$, and
- $n_B \leq n_{\max}$: The maximum distance of relative reference in B should be no larger than $n_{\max}$.

Example 3.3. For example, Table 3.1 shows the concrete tables obtained by applying the standard layout algorithm common among existing ZK compilers to the residual program in Figure 3.6. The standard layout algorithm is simple since it only uses two kinds of *standard gates*: addition ($A + B = C$) and multiplication ($A \times B = C$). Thus, each gate has a maximum degree $d_{\max}$ of 2. Furthermore, it copies every referenced cell to the current row to improve locality, and thus has a maximum relative reference distance $n_{\max}$ of 0.

Table 3.1: Concrete tables obtained from the residual program in Figure 3.6, by applying the standard layout and our layout algorithm, respectively. Each cell is annotated with how it is computed, i.e., either by copying from another cell through absolute reference (e.g., `copy A1`) or by evaluating a polynomial expression over relative references (e.g., $A\langle 0 \rangle * B\langle 0 \rangle$). We use $\overline{var}$ to denote that the cell holds the value of variable `var` in the residual program, while empty boxes denote temporary values.

(a) Standard

	A	B	C	D
0	$\overline{x_0}$	$\overline{z_0}$		0
1	$\overline{sum_0} \leftarrow \text{copy } D_0$	$\overline{i_0} = \text{copy } D_0$		1
2	$\overline{inv_0}$	$\square \leftarrow \text{copy } B_1$	$\square \leftarrow A\langle 0 \rangle * B\langle 0 \rangle$	
3	$\square \leftarrow \text{copy } C_2$	$\square \leftarrow \text{copy } D_1$	$A\langle 0 \rangle = B\langle 0 \rangle$	
4	$\square \leftarrow \text{copy } A_1$	$\square \leftarrow \text{copy } A_2$	$\overline{sum_1} \leftarrow A\langle 0 \rangle + B\langle 0 \rangle$	
5	$\square \leftarrow \text{copy } B_1$	$\square \leftarrow \text{copy } D_1$	$\overline{i_1} \leftarrow A\langle 0 \rangle + B\langle 0 \rangle$	
6	$\overline{inv_1}$	$\square \leftarrow \text{copy } C_5$	$\square \leftarrow A\langle 0 \rangle * B\langle 0 \rangle$	
7	$\square \leftarrow \text{copy } C_6$	$\square \leftarrow \text{copy } D_1$	$A\langle 0 \rangle = B\langle 0 \rangle$	
8	$\square \leftarrow \text{copy } C_4$	$\square \leftarrow \text{copy } A_6$	$\overline{sum_2} \leftarrow A\langle 0 \rangle + B\langle 0 \rangle$	
9	$\square \leftarrow \text{copy } C_5$	$\square \leftarrow \text{copy } D_1$	$\overline{i_2} \leftarrow A\langle 0 \rangle + B\langle 0 \rangle$	
10	$\square \leftarrow \text{copy } C_8$	$\square \leftarrow \text{copy } B_0$	$A\langle 0 \rangle = B\langle 0 \rangle$	

(b) Layout Algorithm

	A
0	$\overline{x_0}$
1	$\overline{z_0}$
2	$\overline{sum_0} \leftarrow 0$
3	$\overline{i_0} \leftarrow 1$
4	$\overline{inv_0} \leftarrow A\langle -1 \rangle * A\langle 0 \rangle = 0$
5	$\overline{sum_1} \leftarrow A\langle -3 \rangle + A\langle -1 \rangle$
6	$\overline{i_1} \leftarrow A\langle -3 \rangle + 1$
7	$\overline{inv_1} \leftarrow A\langle -1 \rangle * A\langle 0 \rangle = 0$
8	$\overline{sum_2} \leftarrow A\langle -3 \rangle + A\langle -1 \rangle$
9	$\overline{i_2} \leftarrow A\langle -3 \rangle + 1$
10	$\square \leftarrow \text{copy } A_1, A\langle 0 \rangle = A\langle -2 \rangle$

3.4.2 General-Purpose Layout Optimization

We give a general-purpose approach for searching an optimal layout using off-the-shelf constraint solvers, by encoding the problem as a MaxSMT instance.

Encoding We start by introducing a *location assignment function* $\alpha : \text{Loc} \times \underline{m} \rightarrow \{0, 1\}$ that relates every location $l \in \text{Loc}$ in the residual program to a set of table's row indices $i \in \underline{m}$. We can then construct a “symbolic” table (denoted by $\tilde{B}$) with each cell yet to be determined:

$$\tilde{B} = \{\langle \text{choose}(g \in G), \text{choose}(i \in \underline{m}) \rangle\}.$$

Here, the **choose** operator indeterministically chooses a value from the provided set. Thus, $\tilde{B}$ encodes all possible table layouts. Then, given t as the residual program, we construct the following hard constraint:

$$\begin{aligned} \forall \gamma_{\text{in}}. (\exists \gamma_t. (\forall l \in \text{Loc}_{\text{in}}. \gamma_t(l) = \gamma_{\text{in}}(l)) \wedge (\gamma_t \models t)) \Leftrightarrow \\ (\exists \gamma_{\tilde{B}}. (\forall l \in \text{Loc}_{\text{in}}. \forall i. \alpha(l, i) \implies \gamma_{\tilde{B}}(i) = \gamma_{\text{in}}(l)) \wedge (\gamma_{\tilde{B}} \models \tilde{B})), \end{aligned}$$

where:

- $\gamma : \text{Loc} \rightarrow \mathbb{F}$ defines a *valuation function* that maps locations to corresponding finite field values. For example, γ_{in} denotes the valuation of user-provided input locations Loc_{in} , γ_t and $\gamma_{\tilde{B}}$ denote the valuation of locations in residual program t and table $\tilde{B}$ respectively.
- The condition “ $\forall l \in \text{Loc}_{\text{in}}. \gamma_t(l) = \gamma_{\text{in}}(l)$ ” restricts γ_t to agree with γ_{in} on all input locations. Similarly, the condition “ $\forall l \in \text{Loc}_{\text{in}}. \forall i. \alpha(l, i) \implies \gamma_{\tilde{B}}(i) = \gamma_{\text{in}}(l)$ ” restricts $\gamma_{\tilde{B}}$ such that all rows i containing some input location l have the same value as $\gamma_{\text{in}}(l)$.
- We use $\gamma_t \models t$ to denote that the constraint system produced from residual program t is satisfied under evaluation γ_t ; as is the case with $\tilde{B}$.

With the soft constraints derived from Equation 3.1, we formulate a MaxSMT instance

that encodes the search for an optimal table layout.

Limitations While the above solver-based general-purpose layout optimization method is guaranteed to find a globally optimal solution (given a finite set of candidate gates G and row indices $\underline{m}$), as we show in our later experiments, it suffers from the following critical limitations:

- The problem search space grows exponentially, regarding the total number of constraints and variables, as the sizes of the table and gates grow.
- Reasoning about highly non-linear finite field operations is notoriously difficult; even state-of-the-art solvers for finite field theory [42] struggle to scale to real-world circuits.

We now introduce an eager layout heuristic that effectively addresses the layout optimization problem in a scalable way.

3.4.3 Layout Algorithm

In this section, we describe an eager layout heuristic for solving the layout optimization problem. As we shall demonstrate in the evaluation, this algorithm achieves better scalability while generating circuits that outperform existing compilers. The pseudo-code is shown in Algorithm 1. The algorithm takes a residual program t as input and constructs a concrete table as output. We assume the existence of procedures $curr_pos()$ and $incr()$ that return the current row position and increment the current row position, respectively.

The algorithm first initializes an empty assignment relation $\alpha : \text{Loc} \times \underline{m}$ that relates each location in the residual program to (a set of) table cells, and an empty selector relation $s : E \times \underline{m}$ that relates each gate expression to (a set of) rows at which the gate is enabled (line 3).

Algorithm 1: Layout Algorithm

```

1 Function layout is
  Input: residual program  $t$ 
2    $\alpha, s \leftarrow \emptyset, \emptyset;$ 
3   for each statement  $c$  in  $t$  do
4     if  $c$  is write  $l \leftarrow \text{read } l'$  then
5        $\alpha[l][\text{last}(\alpha[l'])] \leftarrow \text{true};$ 
6     else if  $c$  is write  $l \leftarrow v$  then
7        $e \leftarrow \text{synth}(v);$ 
8        $p \leftarrow \text{curr\_pos}(); \text{incr}();$ 
9        $\alpha[l][p] \leftarrow \text{true};$ 
10       $s[e - c\langle 0 \rangle][p] \leftarrow \text{true};$ 
11     else if  $c$  is constrain  $v_1 = v_2$  then
12        $e \leftarrow \text{synth}(v_1 - v_2);$ 
13        $p \leftarrow \text{curr\_pos}();$ 
14        $s[e][p] \leftarrow \text{true};$ 
15     end
16 end

```

Then, each statement c in input residual program t is processed sequentially. If the statement is a write operation $\text{write } l \leftarrow \text{read } l'$, the algorithm maps l to the last cell that location l' is assigned to (lines 5-6). If the statement is a generic write operation $\text{write } l \leftarrow v$, the algorithm synthesizes a gate expression e from v by calling *synth*, uses the current cell to hold location l , and enables the gate expression $e - c\langle 0 \rangle$ in the current row (lines 7-11). Finally, if the statement is a constraint operation $\text{constrain } v_1 = v_2$, the algorithm synthesizes a gate expression e for $v_1 - v_2$, and enables it at the current row (lines 12-15). The algorithm then proceeds to the next gate until all gates have been processed.

The *synth* recursive subroutine is shown in Algorithm 2. It pattern-matches on the symbolic value v and synthesizes a polynomial gate expression e that computes v .

- If v reads from a location l , which must have been assigned a concrete cell by some p such that $\alpha[l][p]$ holds, then we first check whether the distance between the current

Algorithm 2: Gate Synthesis

```

1 Function synth is
  | Input: symbolic value  $v$ 
  | Output: gate expression  $e$ 
2 if  $v$  is read  $l$  then
3   | if  $\text{distance}(\text{curr}(), \text{last}(\alpha[l])) > n_{\max}$  then
4   |   |  $\alpha[l][\text{curr\_pos}()] \leftarrow \text{true};$ 
5   |   |  $\text{incr}();$ 
6   | end
7   |  $i \leftarrow \text{diff}(\text{curr\_pos}(), \text{last}(\alpha[l]));$ 
8   | return  $c\langle i \rangle;$ 
9 else if  $v$  is  $v_1 \oplus v_2$  with  $\text{deg}(v) \leq d_{\max}$  then
10  |  $e_1 \leftarrow \text{synth}(v_1); e_2 \leftarrow \text{synth}(v_2);$ 
11  | return  $e_1 \oplus e_2;$ 
12 else if  $v$  is  $v_1 \times v_2$  then
13  |  $e_1 \leftarrow \text{synth}(v_1); p_1 \leftarrow \text{curr\_pos}();$ 
14  |  $s[e_1 - c\langle 0 \rangle][p_1] \leftarrow \text{true}; \text{incr}();$ 
15  |  $e_2 \leftarrow \text{synth}(v_2); p_2 \leftarrow \text{curr\_pos}();$ 
16  |  $s[e_2 - c\langle 0 \rangle][p_2] \leftarrow \text{true}; \text{incr}();$ 
17  |  $p \leftarrow \text{curr\_pos}();$ 
18  | return  $c\langle p_1 - p \rangle \times c\langle p_2 - p \rangle;$ 
19 end

```

cell and the last cell holding l (denoted by $\text{last}(\alpha[l])$) exceeds the maximum allowed relative distance $n_{\max}$ (line 3). If so, we copy l to the current position to improve spatial locality (line 4). We then return the cell reference to the last cell holding l relative to the current row position.

- If v is a binary operation whose degree is no larger than the bound $d_{\max}$, we recursively synthesize the gate expressions for the operands v_1 and v_2 , and return the polynomial expression for the binary operation (lines 9-11).
- Otherwise, we reduce the degree of v by introducing intermediate cells and assigning the operands to these cells (lines 12-18).

3.5 Implementation

TESSEL is written in OCaml (2,811 lines of code), and is connected to the Halo2 [23] backend. Once the program has been compiled into a concrete table, TESSEL performs witness generation given input values to obtain values for all table cells. It then encodes the witness-filled concrete table into the Plonkish Arithmetization Format (PLAF) [80], a standard serialization format for PLONKish circuits.

Witness generation To be able to compute values for all table cells, TESSEL requires a witness computation annotation for each non-deterministic choice in the input program, as standard. The annotated function is preserved throughout the compilation pipeline, and allows the compiler to calculate the value of each concrete cell assigned using an $\star$ gate. The values of other table cells are directly calculated using the assigned polynomial expression.

Code generation We collect the set of all gates in a concrete table, and associate a unique *selector* column s_g to each unique gate expression g . Every polynomial expression gate e enabled at cell $c.r$ is encoded into the polynomial $s_g \times (g - c) = 0$, and the value for s_g is fixed to be 1 at row r . Constraint gates are encoded similarly by multiplying the polynomial with the selector. If gate g is not enabled at cell $c.r$, the selector is set to 0 at row r . The resulting system of polynomial equations and the concrete table values are passed to the Halo2 backend via PLAF to generate the final circuit and proof.

Other optimizations During partial evaluation, TESSEL also attempts to evaluate non-deterministic choices ($\star$) using annotated witness computation functions. If evaluation yields a concrete value, the compiler substitutes it directly rather than introducing a new existential variable. Similarly, constraint operations `constrain  $e_1 = e_2$` are evaluated at

compile-time if both expressions resolve to constants, eliminating unnecessary constraints in the residual program. However, partial evaluation may produce semantically equivalent but syntactically different expressions, such as $x \times 1$ and x , leading to redundant gates and extraneous selector columns. To address this, TESSEL applies **polynomial canonicalization**, normalizing expressions into a consistent coefficient form based on lexicographical variable ordering. Additionally, **constant hoisting** reduces the number of distinct expressions by factoring out constants into fresh variables. For example, expressions like $xy + 1, xy + 2, xy + 3, \dots$ can be compactly rewritten as $xy + z$, where z holds the respective constants. This optimization minimizes the number of gates and selectors, effectively reducing table size with only a slight increase in cell allocation for constants.

3.6 Evaluation

In this section, we describe our evaluation, which aims to address the following research questions:

- **RQ1 (Circuit Quality):** Can TESSEL generate compact circuits?
- **RQ2 (Compiler Performance):** Can TESSEL compile circuits efficiently?
- **RQ3 (Ablation Study):** How does each of TESSEL’s optimization techniques impact the quality of the generated circuits? How does TESSEL perform compared with the solver-based approach with general-purpose SMT encoding?

Benchmarks We sample from 23 open-source repositories written in popular ZK programming languages, such as Noir [81], CirC [27], and Circom [22]. From these repositories, we focus on programs that (1) can be compiled by the Noir/CirC/Circom

compiler itself, and (2) have accompanying test inputs for proof generation (otherwise, we use synthetic test inputs that satisfy the constraints of the original program). As a result, we obtain 26 benchmarks that cover all of the source language constructors shown in Figure 3.4. All of the 26 benchmarks contain loop statements, and 5 of them contain at least one nested loop.

The collected benchmarks span a broad spectrum of application domains, ranging from game logic (e.g., BattleZips, Sudoku) to cryptographic primitives (e.g., modular arithmetic and comparison) and general utilities (e.g., Num2Bits, sort, decoder). Their scale also varies significantly: the smallest kernels consist of only 16 lines of code, whereas the largest arithmetic libraries exceed 173 lines. On average, each benchmark contains around 40 lines. This distribution ensures that our evaluation covers both lightweight programs and computation-heavy routines, thereby providing a representative and challenging testbed for assessing the robustness and efficiency of TESSEL.

Baselines We compare several variants of TESSEL compiler against Noir (0.8.0), CirC (branch zsharp) and Circom (2.2.1) connecting to a Halo2 backend [82]. TESSEL is built upon OCaml version 5.1.1. For all baselines, we target version 0.3.0 of the Halo2 implementation developed by the Privacy and Scaling Explorations team [83] from the Ethereum Foundation [84].

Evaluation metrics To evaluate the effectiveness of TESSEL, we focus on three primary metrics that directly capture the efficiency and compactness of the generated circuits:

- *Proving Time.* This measures the end-to-end time required by the prover to generate a zero-knowledge proof, given a set of inputs and a compiled circuit. Since proving time is often the dominant cost in ZK applications, reductions in this metric translate directly to improved scalability and practicality.

- *Table Size.* This refers to the total number of cells (including input cells, witness cells, and fixed cells) in the instantiated tabular representation. Table size reflects the compactness of the compiled circuit: circuits with fewer rows and columns correspond to fewer constraints, leading to reduced prover workload and smaller memory footprints.

For comparison across compilers, we report results in terms of relative multipliers. Specifically, we normalize all results by setting TESSEL’s performance to 1, and present the baseline systems (e.g., Noir, Circom, CirC) as multiples of TESSEL³.

Evaluation setup Before performing our experiments, we embed all benchmark programs into OCaml in a way that preserves the fine-grained semantics of the original programs. We feed the same inputs to the generated circuits of both TESSEL and Noir/-CirC/Circom. All experiments are conducted on a Macbook Pro laptop with a 12-core Apple M3 Pro CPU and 18GB of memory, running on macOS 14.1.

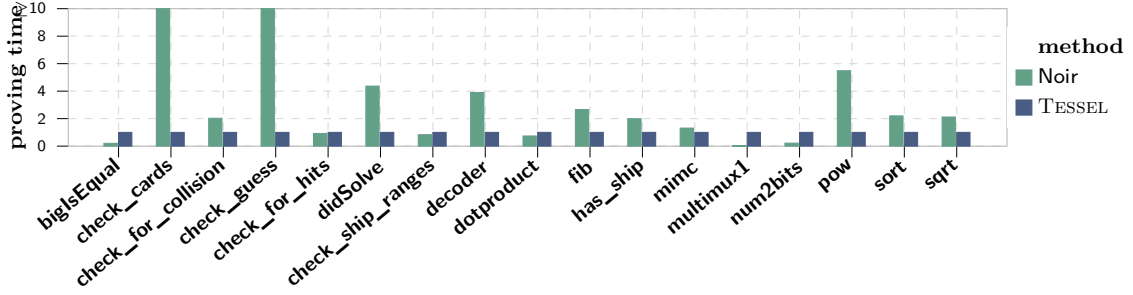
3.6.1 Main Results (RQ1)

We now detail the results for comparison between TESSEL and different popular ZK compilers.

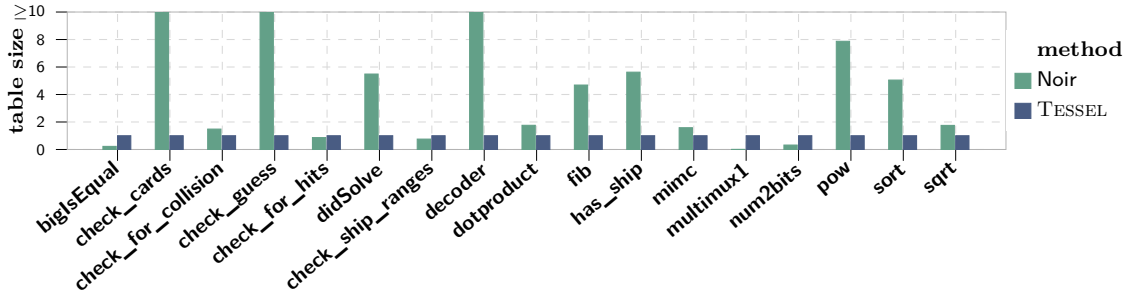
Comparison with Noir

Figure 3.10 presents the comparison of proving time and table size between circuits compiled by Noir and by TESSEL. Overall, we observe that TESSEL achieves substantial improvements in both metrics.

³For example, a proving time multiplier of $5\times$ indicates that the baseline system requires five times longer than TESSEL to generate proofs on the same benchmark. This normalization highlights the performance gap between TESSEL and existing compilers, independent of absolute hardware and implementation differences.



(a) Relative proving time



(b) Relative table size (# of cells)

Figure 3.10: Comparing TESSEL against Noir on proving time and table size. Results are displayed in multiplier of TESSEL.

For proving time, TESSEL yields an average speedup of $2.26\times$, with the maximum of $30.20\times$ on the `check_cards` benchmark (reducing the time from 39.04 seconds to 1.29 seconds). TESSEL reduces the proving time in 13 out of the 16 benchmarks.

For table size, TESSEL reduces the number of cells in 13 out of the 16 benchmarks and by 71.82% on average. The largest reduction is 98.72% on `check_cards`. Importantly, these reductions in table size and proving time do not come at the expense of proof size. In fact, TESSEL consistently produces smaller proofs than Noir across all benchmarks, achieving an average reduction of 65.34%, with the maximum reduction of 80% on `fib` and the minimum of 60% on `check_guess`.

We attribute this performance gap to the fact that the Noir compiler applies only standard compiler optimizations (e.g., constant propagation and common subexpression elimination) before lowering to circuits, whereas TESSEL employs domain-specific symbolic

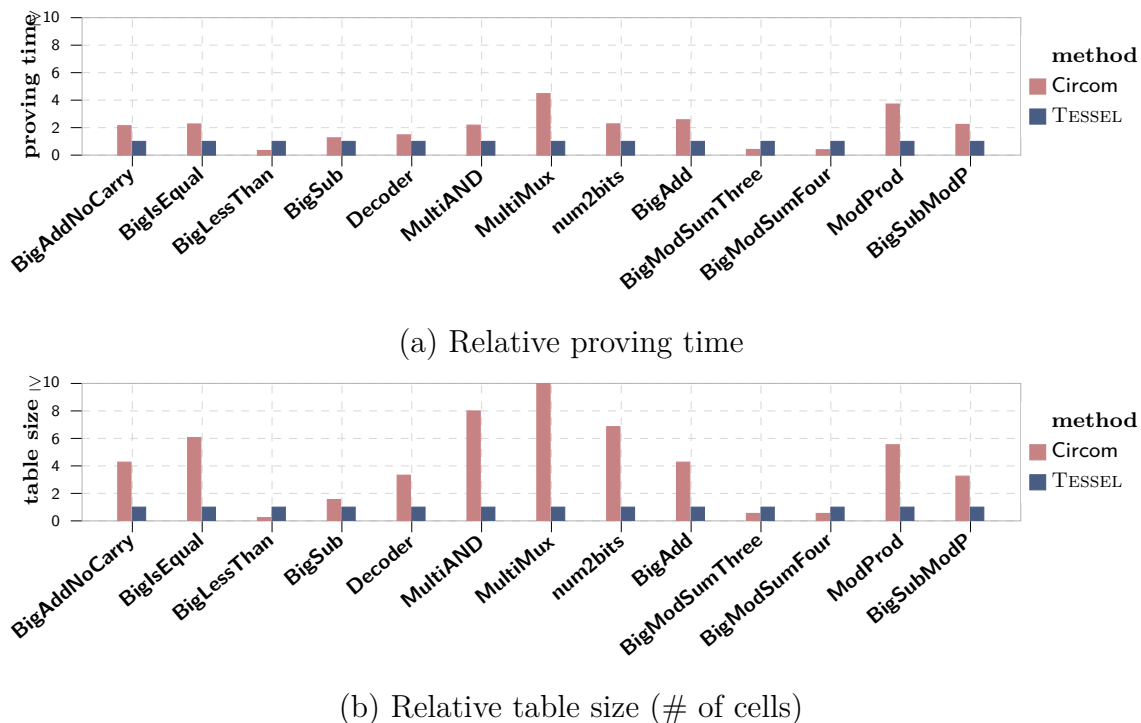


Figure 3.11: Comparing TESSEL against Circom on proving time and table size. Results are displayed in multiplier of TESSEL.

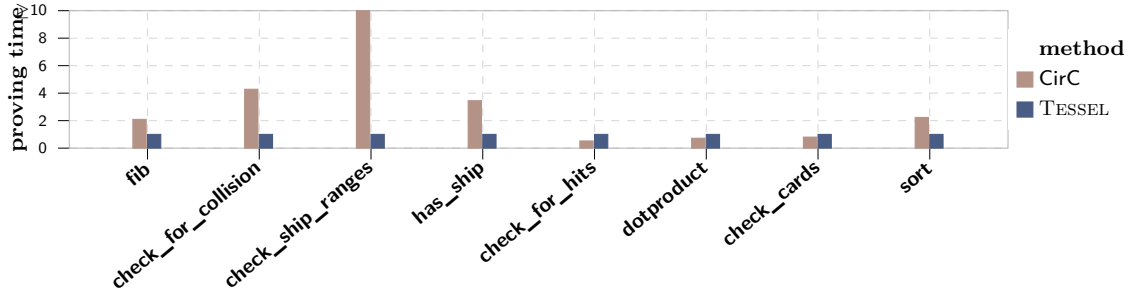
tracing, table IR, and layout optimization techniques that more effectively reduce circuit size.

Comparison with Circom

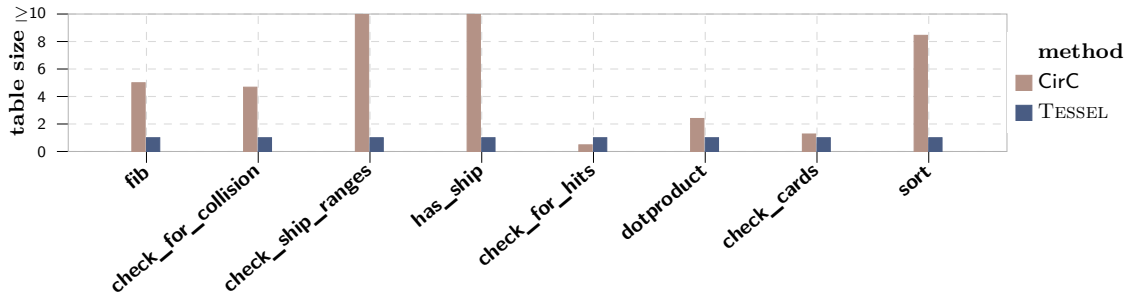
Figure 3.11 reports the results of comparing TESSEL with Circom in terms of proving time and table size. Similar to the comparison with Noir, the results show that TESSEL consistently outperforms Circom across all benchmarks, but with a somewhat different performance profile.

In terms of proving time, TESSEL achieves notable improvements, with an average speedup of $1.81\times$. The most pronounced gain is observed on the **MultiMux** benchmark, where TESSEL reduces proving time from 33.77 seconds to 7.55 seconds (a $4.47\times$ speedup).

For table size, TESSEL also demonstrates significant reductions, averaging 60.82% fewer



(a) Relative proving time



(b) Relative table size (# of cells)

Figure 3.12: Comparing TESSEL against CirC on proving time and table size. Results are displayed in multiplier of TESSEL.

cells than Circom. The maximum reduction reaches 93.12% on **MultiMux**. Importantly, these reductions come without inflating the proof size. TESSEL continues to produce smaller proofs than Circom across most evaluated benchmarks. We also notice that TESSEL increases the table size a bit on two relatively simple benchmarks **BigModSumThree** and **BigModSumFour**.

Overall, these results indicate that while Circom already incorporates domain-specific abstractions for ZK programming, its compilation pipeline does not optimize layout and constraint structure as aggressively as TESSEL. By contrast, the tabular IR and layout strategies in TESSEL enable more compact circuit instantiations and faster proof generation.

Comparison with CirC

Figure 3.12 illustrates the performance comparison between TESSEL and CirC. The results again confirm the advantage of TESSEL, though the magnitude of improvement differs across benchmarks.

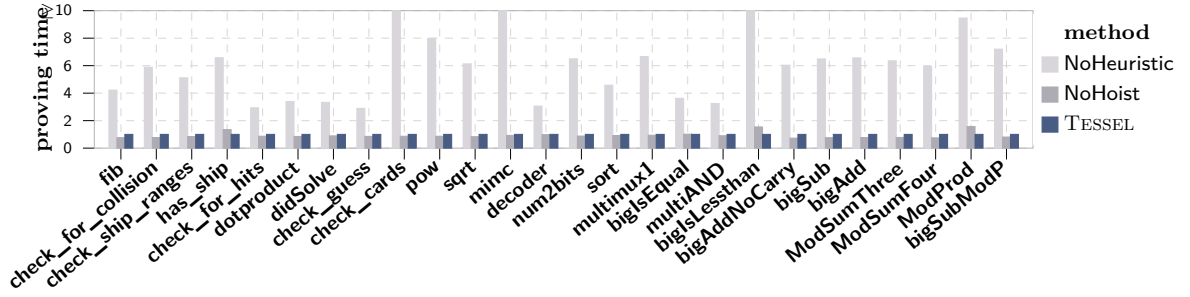
For proving time, TESSEL delivers an average speedup of $4.44\times$. The largest improvement is observed on the `check_ship_ranges` benchmark, where the proving time drops from 56.69 seconds in CirC to only 0.77 seconds in TESSEL, corresponding to a $74.38\times$ acceleration. For table size, TESSEL achieves an average reduction of 89.55%. The most substantial shrinkage occurs on `check_ship_ranges`, where the table size is reduced by 99.58%. Once again, these optimizations are achieved without enlarging proof sizes; in fact, TESSEL consistently produces more compact proofs than CirC.

Taken together, these results show that while CirC offers a flexible compiler infrastructure for multiple proof systems, its general-purpose design leaves optimization opportunities untapped. By contrast, TESSEL’s semantics-guided pipeline focuses on layout efficiency and constraint compaction, resulting in consistently smaller circuits and faster proofs.

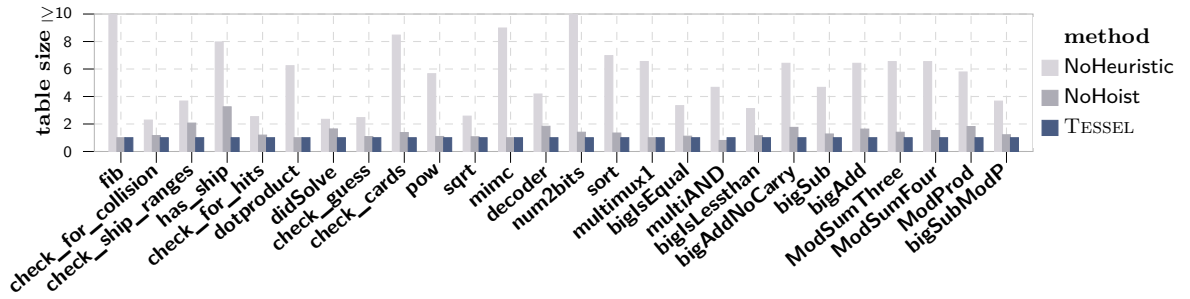
3.6.2 Compilation Efficiency (RQ2)

We evaluate the compilation efficiency of TESSEL against Noir, Circom, and CirC. For Noir, compilation time is measured as the duration to translate a program into ACIR via `nargo`; for Circom and CirC, it is the time to lower a program to their respective circuit representations; for TESSEL, it is the time required to generate the table IR with witness values.

Compared to Noir, TESSEL compiles substantially faster, with an average of 0.52s (minimum 0.27, maximum 1.56) versus Noir’s 10.43s (minimum 1.03, maximum 37.58).



(a) Relative proving time



(b) Relative table size (# of cells)

Figure 3.13: Comparison of key metrics (proving time and table size) among variants of TESSEL. Results are displayed in multiplier of TESSEL’s full-fledged version.

This shows that the additional symbolic tracing in TESSEL does not hinder, and in fact improves, end-to-end compilation efficiency. When compared with CirC, TESSEL also compiles slightly faster (0.61s vs. 0.96s).

Relative to Circom, TESSEL incurs higher compilation times on average (2.73s vs. 1.36s). We attribute this to Circom’s lightweight front-end translation, which avoids complex optimizations but results in larger circuits and longer proving times downstream.

In summary, while TESSEL’s compilation overhead is somewhat higher than Circom, it remains practical and is amply justified by the significant reductions in proving time and proof size observed in our evaluation.

3.6.3 Ablation Study (RQ3)

To find out the impact of each of TESSEL’s optimization techniques, we now elaborate the results of the comparison between different ablative versions of TESSEL and a solver-based baseline.

Comparison with Different Ablative Versions

We conduct experiments to study the effectiveness of the proposed optimization, using two variants of TESSEL. The first variant, denoted by **NoHeuristic**, is obtained by replacing the eager layout heuristic with a standard layout algorithm that uses two kinds of gates – addition and multiplication. The second variant, denoted by **NoHoist**, is obtained by removing the optimization of polynomial canonicalization and constant hoisting. We refer to the original full-fledged version as TESSEL.

The comparison results are shown in Figure 3.13. We first discuss the comparison between TESSEL and **NoHeuristic**. Overall, the layout algorithm significantly reduces the proof generation time, while slightly increasing the average proof size. The average speedup of TESSEL over **NoHeuristic** in proving time is $6.92\times$, with a maximum speedup of $10.99\times$ (on benchmark `check_cards`) and a minimum of $3.04\times$ (on benchmark `decoder`). Meanwhile, the layout optimization also effectively reduces the table size. Enabling the algorithm decreases the table sizes on average by 76.10%. In terms of proof size, TESSEL incurs a slightly higher cost, as **NoHeuristic** generates proofs that are on average 33% smaller. This is explained by the inherent cost of TESSEL having to commit to a larger set of synthesized custom gates than just the standard gates (**NoHeuristic**). This shows that the layout algorithm makes a better trade-off between proof generation time and proof size compared to the standard layout algorithm.

Next, we compare TESSEL against **NoHoist**. It can be observed that enabling the

optimizations of polynomial canonicalization and constant hoisting (**NoHoist**) results in a decrease in table size in many benchmarks. On average, this optimization achieves a 24.5% reduction in table size. The reason for this reduction is that, by eliminating constant terms and normalizing the polynomial expressions at compile time, the polynomial gates of the circuit are simplified, resulting in a smaller set of gates, and thus a smaller table size. Moreover, constant hoisting consolidates similar polynomials and further contributes to the reduction.

Comparison with Solver-Based Approach

We compare the performance of our algorithm against the baseline encoding (Section 3.4.2), which leverages the Z3 [85] solver to solve constraints. Since the baseline approach struggles to scale with most of our benchmarks, we use Fibonacci, the simplest benchmark in our dataset, as a case study. As the proof size increases with larger input parameters, we gradually increase the argument from 1 to progressively larger values. The baseline encoding fails to complete within 30 minutes when $N \geq 10$, whereas our layout algorithm efficiently solves the same benchmark in 0.223 seconds. This result highlights the scalability of our approach, demonstrating that it can handle significantly larger problem instances compared to the naive solver-based method.

3.6.4 Threats to Validity

This section outlines possible limitations of our study and their implications for the results.

Internal Results could be distorted by defects in either TESSEL or our measurement harness. We mitigate this risk by (i) verifying every circuit produced by TESSEL against Noir’s reference compiler on a common Halo2 backend, and (ii) executing all experi-

ments through fully scripted pipelines on an isolated, homogeneous workstation, thereby eliminating manual intervention and hardware variability.

Metrics Our study focuses on three metrics—proving time, proof size, and table size—because they directly capture the resource costs that layout-level optimizations aim to reduce. Metrics less sensitive to layout (e.g., verifier latency) are intentionally excluded to maintain a clear link between the intervention and the observations.

External The benchmark suite comprises cryptographic primitives, numerical kernels, and application-level Noir programs that exercise the full language surface used in practice. Although empirical behaviour may vary for proof systems other than PLONK-style or for highly specialised hand-written circuits, TESSEL’s intermediate representation and algorithm are proof-system agnostic, suggesting that the qualitative findings generalise beyond the evaluated workloads.

3.7 Related Work

In this section, we examine the most closely related works relevant to our proposed approach.

Zero-knowledge proof systems and arithmetizations zk-SNARK [61] is one of the prevailing ZK proof systems used in real-world privacy-aware application domains, such as blockchains and authentication systems, where anonymous transactions and trustless verification are indispensable. There are many realizations and variants of the zk-SNARK proof system, such as Groth16 [9] and PLONK [10], where Groth16 is based on R1CS arithmetization and PLONK is based on PLONKish arithmetization, which is considered a more general class of polynomial equations over finite fields. Variants of

PLONK (e.g., HyperPLONK [11], TurboPLONK [12] and UltraPLONK [13]) share the same PLONK-style arithmetization, with compatible extensions, which are all referred to as PLONKish arithmetization. As PLONKish arithmetization is a super class of R1CS, the core techniques of TESSEL also apply to other arithmetization schemes like R1CS. zk-STARK [75] is one of the other popular ZK proof systems, whose typical realizations such as Cairo [15] base themselves on AIR arithmetization. Since AIR is usually considered a restricted form of PLONKish arithmetization, TESSEL is also applicable.

ZK languages and compilers Languages specifically designed for writing ZK programs have been a popular topic for both research and industrial work. They all compile ZK programs into their corresponding ZK components, but with a different focus. Languages like Circom [22], ZoKrates [26] and Noir [24] focus on providing a higher-level domain-specific interface that abstracts away the tedious and error-prone ZK constraint manipulation, leaving the developers with a set of general-purpose operators for describing their computation. Other languages like Leo [25], Cairo [15] and zkEVM [86, 17] elaborate more on the close binding of ZK language constructs with blockchain platforms (such as Ethereum and Starknet), where more complicated operations are packed into APIs. While many of those languages are built upon Halo2 [23], rather than stacking on top of it, the most recent ZK languages instead seek a “horizontal” extension by creating ZK virtual machines for general-purpose programming languages. For example, RISC Zero [28], SP1 [29], and Jolt [30] “decorate” Rust constructs with their corresponding ZK constraints. This shortens the compilation workflow and makes the ZK compiler behind the virtual machine and its optimization a critical and challenging topic.

ZK intermediate representations and compiler optimization Arithmetization is one kind of intermediate representations (IRs) that many ZK compilers directly build

upon. For example, Circom [22] compiles its programs into R1CS, and performs constraint-level optimization [16] over it. Leo [25] programs get compiled into an IR called Aleo before converting into R1CS. As Aleo preserves high-level semantics, typical compiler optimization techniques such as constant folding and propagation, loop unrolling and function inlining, can all be applied on application level. There are also improvements and architectures that support compiler optimization, such as plookup [87] and CirC [27].

There are also dedicated IRs that enable compiler optimization toward different goals. For example, Noir [24], Chiquito [88] and Juvix [89] are designed for better usability as well as compatibility with specific blockchains. CODA [53] is designed for composing safe ZK circuits. CirC [27] focuses more on compatibility between different ZK languages. As the IRs mentioned focus more on usability and language compatibility, TESSEL demonstrates a focus on performance optimization for ZK circuits.

3.8 Conclusion

This chapter introduces TESSEL, a compiler that bridges the gap between programmability and efficiency in zero-knowledge (ZK) development. By optimizing the translation of high-level ZK programs into low-level circuits, TESSEL leverages partial evaluation, symbolic tracing, and a layout algorithm to reduce circuit size and enhance performance. Our results show that TESSEL achieves a 1.81–4.44× speedup over mainstream compilers.

Chapter 4

Tabby: A Synthesis-Aided Compiler for High-Performance Zero-Knowledge Proof Circuits

4.1 Introduction

In this chapter, we continue our journey to improve the efficiency of ZK circuits compiled from high-level DSLs. Previously, in Chapter 3, we formulated the optimization of PLONKish circuits as a precisely defined *table-layout problem* and tackled it with an eager refinement heuristic that aims to reduce proving time. Because the algorithm is heuristic in nature, however, the improvements it produces are not guaranteed: on some inputs the heuristic can in fact *enlarge* the resulting table and proof, since the local decisions it commits to early on are never revisited. This chapter presents TABBY, a

Adapted, with permission, from Junrui Liu, Jiaxin Song, Yanning Chen, Hanzhi Liu, Hongbo Wen, Luke Pearson, Yanju Chen, and Yu Feng, “Tabby: A Synthesis-Aided Compiler for High-Performance Zero-Knowledge Proof Circuits,” *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA, October 2025. © 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM. Version of Record: <https://doi.org/10.1145/3763110>.

compiler that tackles the same circuit-optimization problem formalized in Chapter 3 but reframes it as a *program synthesis* problem. By searching the space of equivalent circuits rather than eagerly reducing a single candidate, TABBY yields more consistent reductions in proving time across a much wider range of inputs.

The key insight of TABBY compared to other high-level DSL compilers (e.g., TESSEL, Leo, Noir) is as follows. Existing compilers translate circuits from high-level DSLs to a predefined, universal constraint template called *standard gates* to encode the computation to be proven. As a result, for complex operations—especially those involving repeating computation patterns—existing compilers natively unroll the loops, leading to a large number of statements and variables. Each statement may require a large number of standard gates to represent, increasing the circuit size and consequently the proving time. To address this problem, domain experts bypass the compilers and manually fine-tune low-level circuits by designing *custom gates* that define specialized polynomial constraints tailored to specific and recurring computational patterns within their circuits. By aligning the gate design with the frequently occurring computation patterns, developers can exploit the inherent structures within the problem, reducing the number of gates and constraints needed.

Although designing custom gates by hand can significantly reduce the circuit size, the process is non-trivial: developers must (1) understand the original computation and (2) design custom gates that lead to equivalent computations while minimizing the number of polynomial constraints. One alternative to manual fine-tuning is to leverage super-optimization [90] to automatically search for alternatives that can improve the existing circuit while preserving its semantics. However, this straightforward strategy is infeasible in this domain: a circuit can contain millions of gates, and a gate can be an arbitrary polynomial expression over field elements. Traditional super-optimization techniques only scale to code of a few dozen instructions [91], making them unsuitable for

handling the complexity of ZKP circuits.

Instead, we introduce TABBY, a compiler for generating high-performance ZKP circuits that solves the same optimization problem formalized in Chapter 3 but reframes it as program synthesis. TABBY allows developers to have the best of both worlds by abstracting the complexity of optimizing low-level circuits without incurring significant performance overhead. Our key insight is that, instead of framing the problem as super-optimization at the gate level, we propose a circuit intermediate representation that exposes crucial abstractions that domain experts typically use during optimization. For instance, our DSL naturally supports important constructs and operations in mainstream ZKP systems, such as gadgets for representing low-level custom gates. Second, unlike existing ZKP compilers that only perform standard compiler optimizations, TABBY leverages program synthesis [48, 49, 50] to search for a new circuit that (a) is equivalent to the original one and (b) leverages custom gates to reduce the proving time. Because synthesis explores the space of equivalent programs rather than eagerly mutating a single one, TABBY avoids the local-minimum pitfall of the heuristic in Chapter 3 and consistently improves over it on the same benchmark suite. Finally, as we show in our evaluation, even with the aforementioned reduction, synthesizing high-performance circuits remains challenging due to the large search space. To make our synthesis procedure scalable to real-world tasks, we employ a goal-directed approach where the search is guided by the abstract semantics of the original program (i.e., reference implementation). TABBY also leverages the data dependencies of the original task to decompose a complex synthesis problem into smaller sub-tasks.

We evaluate TABBY on a diverse suite of real-world ZKP programs drawn from popular libraries and application benchmarks. Our results show that TABBY consistently produces more efficient arithmetic circuits compared to mainstream ZKP compilers [22, 36] and to the heuristic of Chapter 3, yielding substantial reductions in proof generation time.

To demonstrate its generality across proving systems, we also adapt TABBY to target a STARK-based backend (Plonky3 [92]), where it achieves significant performance gains in a case study, including up to $50\times$ improvement in proving time.

In summary, this chapter makes the following contributions:

- We present TABBY, a compiler for zero-knowledge proof circuits that balances programmability and efficiency.
- TABBY introduces a circuit intermediate representation exposing crucial abstractions like gadgets to aid optimization.
- TABBY reformulates the circuit-optimization problem of Chapter 3 as program synthesis, avoiding the local-minimum pitfalls of an eager heuristic by searching the space of equivalent circuits.
- TABBY uses program synthesis to generate equivalent circuits that leverage custom gates, significantly reducing proof generation time.
- TABBY outperforms mainstream compilers by generating 70% smaller circuits.

4.2 Background and Overview

In this section, we provide some necessary background on zero-knowledge proofs, followed by a real-world ZKP circuit written in Noir [36] to explain the TABBY workflow.

4.2.1 Background

Zero-knowledge proofs ZKPs enable a prover ($\mathcal{P}$) to convince a verifier ($\mathcal{V}$) that they possess certain private data without revealing the data itself. A compelling example involves a computational process $\mathcal{C}$ with public inputs (x) and private inputs (y). The

prover’s goal is to convince the verifier that the output z is the result of $C(x, y)$, without revealing anything about y . ZKPs ensure that the prover can demonstrate knowledge of y without disclosing any information about it. Among various ZKP protocols, zkSNARKs stand out for their efficiency, offering compact proofs and fast verification times. One of their key advantages is the ability to automatically generate a prover and verifier from an arithmetic circuit that represents the computation.

Noir [36] is a domain-specific language designed for creating and verifying zero-knowledge proofs using SNARKs. It streamlines ZKP development with a syntax inspired by Rust and abstracts the underlying cryptographic complexities. Noir supports familiar programming constructs like functions, conditionals, loops, and user-defined types, making it accessible and intuitive for developers, particularly those with Rust experience.

Figure 4.1 (A) shows a simple example in Noir [36]. In particular, Noir abstracts the underlying cryptographic complexities, allowing developers to focus on the logic of their applications. Noir’s design allows it to be compatible with any Abstract Circuit Intermediate Representation (ACIR) compatible proving system, such as PLONK [10], Groth16 [9], and Marlin [93]. This flexibility makes Noir a powerful tool for developers working on zkSNARK-based applications.

Proof systems, R1CS, and PLONK In the world of zero-knowledge proofs, proof systems are used to reduce arithmetic circuits to zero-knowledge proofs that ensure the circuit’s satisfiability, given some input. Popular arithmetic circuit types include R1CS [55] and PLONKish [10, 92, 11].

An R1CS (short for Rank-1 Constraint System) can be used to translate a computational problem into a system of linear equations. More concretely, an R1CS is a relation $\Re(\vec{z}^n, A^{n \times k}, B^{n \times k}, C^{n \times k})$, and it represents the computation as k linear combinations, expressed by a vector of inputs and witnesses, and three matrices. At each step, we apply

only a single gate, which roughly looks like this: $\sum_{i \in [k]} A_i \vec{z}^n \cdot \sum_{i \in [k]} B_i \vec{z}^n - \sum_{i \in [k]} C_i \vec{z}^n = 0$. So at each step, we take the row i of the matrices A, B, C and multiply each row individually with $\vec{z}^n$. Thus, the A, B, C matrices encode the usage of values in the input and witness vector at a given gate. Let's go through an example: $x^3 + 2x + 4 = 0$. To reduce this to an R1CS relation, first, we need to flatten the program into a series of steps that fit the constraint equation.

$$(\text{sym}_1 = x \times x, \quad \text{sym}_2 = \text{sym}_1 \times x, \quad \text{sym}_3 = 2 \times x, \quad \text{out} = \text{sym}_2 + \text{sym}_3 + 4).$$

This now gives us the names of all variables we will use in the R1CS inputs and witness vectors. Thus, $\vec{z}^n$ will look like $[1, x, \text{out}, \text{sym1}, \text{sym2}, \text{sym3}]$ and $n = 6$. Note that the “1” here is used for constants in the computation, which will become clear in a moment. Next, we need to create our “usage” matrices, named **execution trace table**, to track the evolution of variables and collect intermediate values to construct the proof. Taking our previously created vector $\vec{z}^n$, we can do this easily step by step:

1. First, we need to multiply x with x , and it should equal sym1 . So, the first row of A and B have element ‘1’ corresponding to x in $\vec{z}^n$.
2. Next, we are multiplying sym1 with x , and the output should equal sym2 . Thus, the next row of C has element ‘1’ corresponding to sym2 :
3. After the third and final step, we complete the relation $\Re(\vec{z}^n, A^{nxk}, B^{nxk}, C^{nxk})$ as:

$$\vec{z}^n = [1, x, \text{out}, \text{sym1}, \text{sym2}, \text{sym3}].$$

$$\begin{aligned} A_0 &= [0, 1, 0, 0, 0, 0]; & A_1 &= [0, 0, 0, 1, 0, 0]; & A_2 &= [2, 0, 0, 0, 0, 0]; & A_3 &= [4, 0, 0, 0, 1, 1] \\ B_0 &= [0, 1, 0, 0, 0, 0]; & B_1 &= [0, 1, 0, 0, 0, 0]; & B_2 &= [0, 1, 0, 0, 0, 0]; & B_3 &= [0, 0, 0, 0, 0, 0] \\ C_0 &= [0, 0, 0, 1, 0, 0]; & C_1 &= [0, 0, 0, 0, 1, 0]; & C_2 &= [0, 0, 0, 0, 0, 1]; & C_3 &= [0, 0, 1, 0, 0, 0]. \end{aligned}$$

An R1CS-based proof system can then take this relation and produce a zero-knowledge proof of it with a given input for x .

PLONKish allows more complex polynomial constraints than R1CS, providing greater efficiency and flexibility. For instance, Plonky2, a SNARK implementation based on techniques from PLONK and used by Halo2 and Noir, organizes the execution trace table into three types of columns:

- **Fixed columns:** Contain constants that remain unchanged throughout the computation and when the circuit is instantiated with different input values.
- **Instance columns:** Store public input values that are part of the statement being proved.
- **Advice columns:** Hold witness data, including intermediate and final results of the computation.

The execution trace table captures all necessary data for constructing and verifying proofs. It ensures the integrity and correctness of the computation by allowing each step to be checked against the defined polynomial constraints.

Custom gates In proof systems like PLONK and its derivatives, circuits are constructed using polynomial constraints that encode the computation to be proven. While basic, predefined constraints (standard gates) are sufficient for representing simple arithmetic operations, they can be inefficient for complex computations. This inefficiency arises because complex operations may require multiple standard gates, increasing the circuit size and, consequently, the proving time.

Custom gates address the limitations of standard gates by allowing developers to define specialized polynomial constraints tailored to specific computational patterns within their circuits. By aligning gate design with the specific arithmetic patterns of

the computation, developers can exploit inherent structures within the problem, leading to more compact and performant circuits. By encapsulating complex operations into single, higher-degree polynomial equations, custom gates reduce the number of gates and constraints needed, significantly decreasing both the size of the circuit and the time required for proof generation and verification. An important insight is that custom gates enable optimizations unattainable with standard gates, making them essential for high-performance ZKP applications.

Signal rotation Signal rotation is used in proof systems to express constraints between a signal’s values at different positions in the execution trace. Specifically, it allows a circuit to relate the value of a signal at one step to its value at another step, effectively shifting the reference point within the trace. For shared signals consistent across steps, constraints can be imposed on their values at different relative positions. For example, if a is a shared signal, one might enforce that the value of a at the current step equals its value two steps ahead in the execution trace. Signal rotation enables the expression of temporal relationships within the trace, which is particularly useful for encoding iterative processes and recurrences compactly within the circuit.

Circuit optimization Optimizing circuits can significantly reduce *prover time* when generating proofs, making ZKP-based applications more practical and scalable for broader use. Improvements in circuit efficiency directly impact performance and scalability in many privacy-focused applications such as blockchain. Standard optimization techniques, such as eliminating redundant operations and combining multiple operations into more efficient ones, are widely adopted. However, the most effective optimizations are circuit-specific. Reducing circuit depth, minimizing constraints, optimizing the use of arithmetic gates, and improving the execution trace table can further decrease computational complexity.

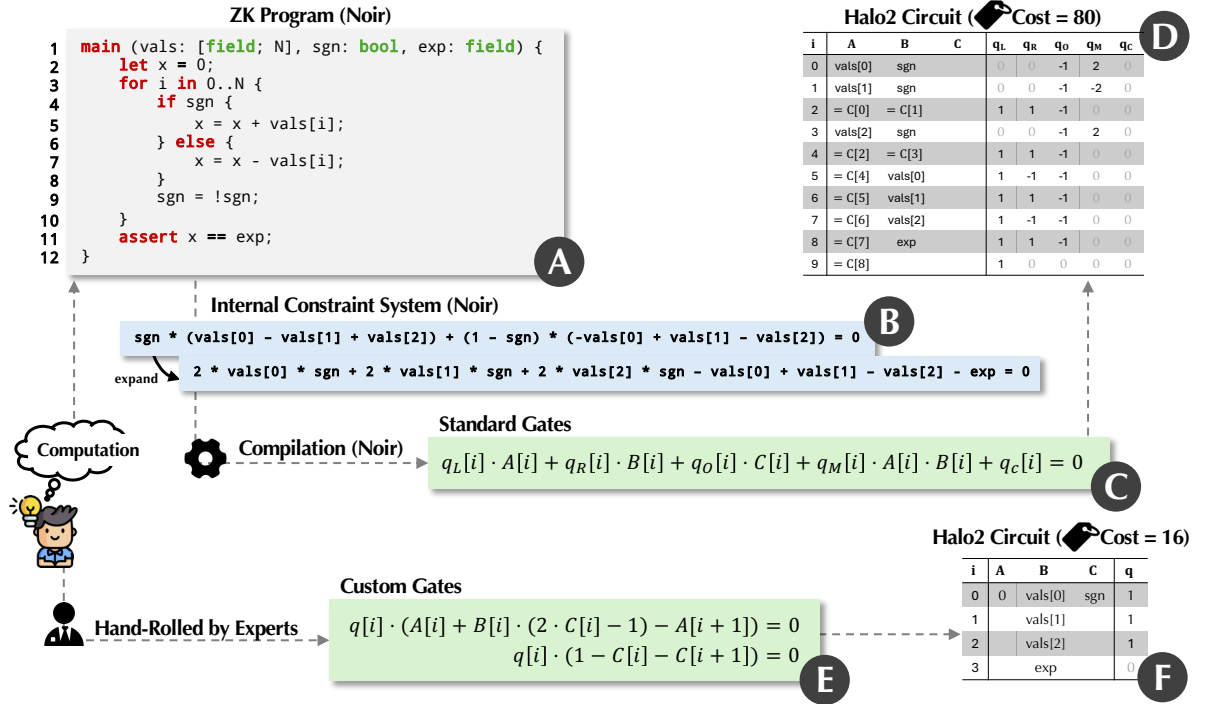


Figure 4.1: Illustration showing how a circuit can be encoded as a trace table via standard gates for the Halo2 proving system ($A \rightarrow B \rightarrow C \rightarrow D$). In order to optimize the size and performance of the circuit, experts write custom gates to dramatically shrink the size of the generated trace table ($A \rightarrow E \rightarrow F$), which requires non-trivial efforts.

Efficient data structures that facilitate quicker access and manipulation during circuit execution are also essential.

4.2.2 System Overview

In what follows, we give an overview of the TABBY system, the insights behind it, and how it works to automatically optimize a circuit via program synthesis.

Circuit encoding As circuit optimization is important for real-world applications, having a reliable and effective procedure is non-trivial as many circuits require expert knowledge to apply valid optimization. We show an example circuit and its optimization in Figure 4.1. Particularly, given an array `vals`, the original circuit **A** subtracts the sum of

all its elements at odd (resp. even) positions from that at even (resp. odd) positions if the initial `sgn` is true (resp. false). In the `main` function, variable `x` is used as the final result, and within the `for` loop that performs the accumulation, `sgn` is used as an alternating operator that adds (resp. subtracts) the current array elements to (resp. from) variable `x`.

In order for downstream cryptographic libraries to consume and process the circuit, prevailing ZK programming languages and compilers (such as Noir [36] and Circom [22]) need to first convert the circuit into a *structured* math object that encodes *equivalent* semantic meaning of the circuit (in its internal representation as shown in **B**). Figure 4.1 **D** shows such an object — a table — that is used by one of the popular ZK proving systems called Halo2 [84]. In particular, each row of the table represents a single step of the circuit’s execution trace with the help of a special structure called *gate*. To see how, we show the *standard gate* used in default ZK compilation in Figure 4.1 **C**:

$$q_L[i] \cdot A[i] + q_R[i] \cdot B[i] + q_O[i] \cdot C[i] + q_M[i] \cdot A[i] \cdot B[i] + q_c[i] = 0,$$

where q_L, q_R, q_O, q_M and q_C are fixed columns. The standard gate can be viewed as a *template* for building constraints: one simply needs to choose appropriate values for the fixed columns to express the desired polynomial operations. The remaining columns A , B and C will be filled with instance (or witness) values by the prover. For example, if we plug in the values from each cell of the table’s first row into the above gate, we then get the following concrete constraint:

$$0 \cdot \text{vals}[0] + 0 \cdot \text{sgn} + (-1) \cdot C[i] + 2 \cdot \text{vals}[0] \cdot \text{sgn} + 0 = 0, \quad (4.1)$$

which evaluates to $2 \cdot \text{vals}[0] \cdot \text{sgn} = C[i]$.

Equation 4.1 corresponds to the first term of the expanded version of the internal constraint system **B**. When proving the correctness of the execution, one needs to submit the actual

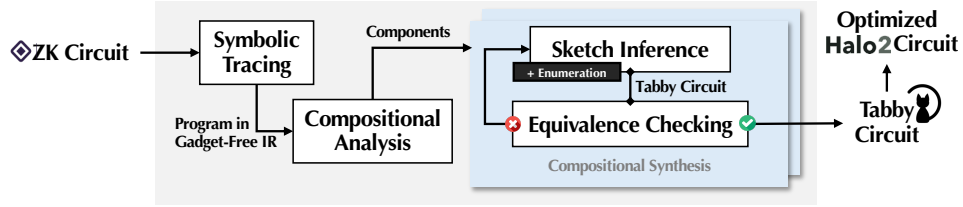


Figure 4.2: An overview of TABBY.

value of x during execution, which is then plugged into Equation 4.1. Only when the result checks out, for example:

$$2 \cdot \text{vals}[0] \cdot \text{sgn} = C[i], \text{ where e.g., } \text{vals}[0] = 2, \text{sgn} = 1, C[i] = 4 \quad (4.2)$$

will the current term be proved correct. Therefore, by converting the entire execution trace of the circuit using the standard gate we end up with the trace table in Figure 4.1 **D**, which corresponds to a set of constraints that encodes the expected correct execution trace of the circuit.

Challenges for circuit optimization In practice, the performance of a circuit is heavily affected by the shape, and especially the size of its corresponding trace table. The trace table shown in Figure 4.1 **D** contains a total of 80 cells (excluding the first indexing column), which is much larger than one of the minimum possible encoding as shown in Figure 4.1 **F** that has only 16 cells. In order to achieve such a significantly smaller-sized trace table, it usually requires additional help from the so-called *custom gates* — a more flexible form of constraint templates — as shown in Figure 4.1 **E**. However, writing custom gates that could 1) successfully encode the original circuit, and 2) reduce the size of the trace table, is non-trivial and heavily requires expert knowledge. It took a well-known team with two cryptography experts six months to derive an optimized version of a ZK library for big integer arithmetics, which only contains 20 utility functions [94].

As the compilation from the original program to the trace table has a concrete requirement (i.e., semantic equivalence) and objective (i.e., smaller-sized trace table), an optimal synthesis algorithm can potentially help reduce the amount of expertise and manual effort required for circuit optimization. However, directly searching for an equivalent set of custom gates that minimize the size of the trace tables can quickly become intractable, as realistic circuits may have tables with well over millions of rows [4], and there are potentially infinitely many configurations of custom gates that express the same computation.

The key insight of TABBY is to constrain the space of synthesis search to an intermediate representation (IR) which we call $\mathcal{L}_{\text{IR}}$. The design of $\mathcal{L}_{\text{IR}}$ specifically proposes the *gadget* abstraction to capture the set of computation patterns for which compiling to custom gates (instead of standard gates) provides the most efficiency gains, as distilled from existing manual optimizations performed by expert developers. Because expressing those computation patterns using custom gates already provides substantial performance gains in the compiled circuits, TABBY merely has to search for an *equivalent* gadget representation, thus avoiding the need for an expensive optimal synthesis search.

Framework overview As shown by Figure 4.2, TABBY takes as input a circuit written in ZK programming languages, and searches for its best configuration of custom gates that results in better performance; TABBY returns a semantically equivalent circuit written in an intermediate representation language called $\mathcal{L}_{\text{IR}}$, which can be compiled to a Halo2 circuit that has an optimized trace table.

From within TABBY, a compositional synthesis procedure is invoked to search for the optimized version of the given circuit. In particular, TABBY starts with a symbolic tracing procedure that pre-processes the original circuit into its internal representation (IR). Then TABBY runs a compositional analysis procedure that identifies different independent

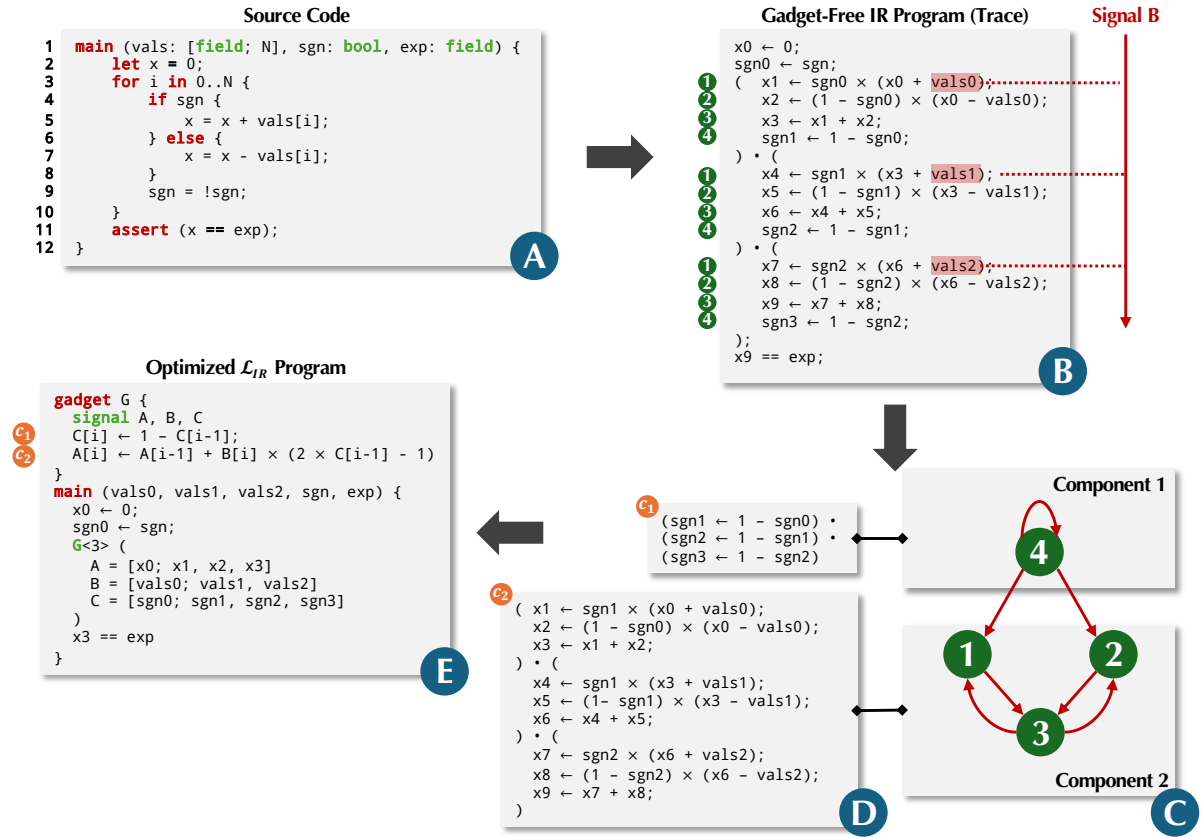


Figure 4.3: A walk-through example showing how TABBY finds an optimized version of a given circuit.

components of the circuit, where each component corresponds to an IR snippet. For each component, TABBY then invokes a sketched-based program synthesis procedure to search for its equivalent counterpart written in $\mathcal{L}_{IR}$ with optimized complexity. Finally, by assembly of the $\mathcal{L}_{IR}$ snippets, TABBY produces an optimized version of the circuit.

A walk-through example We show with an example how TABBY converts a circuit into its optimized version in $\mathcal{L}_{IR}$ in Figure 4.3, which breaks down to the following steps:

- Starting with the source code in Figure 4.3 (A), TABBY evaluates it into a program written in its intermediate representation, i.e., a trace program, as shown in Figure 4.3 (B). The trace program gets rid of some high-level language structures such as loops

by unrolling them. As loops in ZK circuits should always be bounded as required from the underlying cryptographic design, the trace program should completely unroll all iterations of a loop.

- After that, TABBY identifies patterns of computation from the trace program; for example, ①②③④ is a repetitive pattern for value accumulation that appears multiple times in the trace program, which could be “summarized” as a custom gate as it essentially describes a constraint template that can be reused for each repetition.
- TABBY performs a *component analysis* on the identified pattern and generates a dependence graph for each statement within, as shown by Figure 4.3 ③. The dependence graph is further decomposed into smaller units called components in a deterministic procedure, such that each component itself can be viewed as an independent code snippet for optimization. The corresponding trace program for each component is shown in Figure 4.3 ④.
- Then for each component, TABBY invokes a sketch-based program synthesis algorithm that searches for its equivalent version written in the proposed $\mathcal{L}_{\text{IR}}$ language. As shown in Figure 4.3 ⑤, TABBY introduces a language construct called *gadget*, which encapsulates the two synthesized custom gates ⑥₁ and ⑥₂ which correspond to the pattern observed in ④. The gadget G is expanded in the `main` program that produces corresponding constraints.
- Through TABBY’s backend, the $\mathcal{L}_{\text{IR}}$ program gets further compiled to a Halo2 circuit whose trace table is then reduced to the size of 16, which is much smaller than the ones compiled with the default configuration of gates.

$e ::=$		Expression:
$f \in \mathbb{F}$		field constant
x		variable
$e \oplus e$		binary operation
$x[e]$		array read
$c ::=$		Statement:
$x := e$		assignment
$x[e] := e$		array write
$c; c$		sequencing
repeat $e \{c\}$		loop
if e then c else c join		conditional
constrain $e = e$		constraint
$\oplus \in$	$\{+, -, \times\}$	
$p ::= g^* m$		Program
$m ::= \cdot \mid m; s$		Statement
$m; t$		Loop trace
$m; G\langle n \rangle((S = \sigma)^*)$		Gadget expansion
$s ::=$		Statement
$x \leftarrow e$		Variable write
$e = e$		Constraint
$S[i + \delta] \leftarrow e$		Signal assignment
$e ::=$		Expression
$f \in \mathbb{F}$		Field constant
$e \oplus e$		Binary operation
$S[i + \delta]$		Signal rotation
x		Variable
$t ::= \cdot \mid s^* \bullet t$		Loop trace
$g ::= \text{gadget } G \{S^*; s^*\}$		Gadget
$\sigma ::= [x^*; x^*]$		Signal literal
$\delta \in$	$\mathbb{Z} = \{0, 1, -1, 2, -2, \dots\}$	
$n \in$	$\mathbb{N} = \{0, 1, 2, \dots\}$	

Figure 4.4: Syntax of $\mathcal{L}_{\text{ZK}}$ (top) and syntax of the intermediate representation $\mathcal{L}_{\text{IR}}$ (bottom). Constructs colored in blue can only appear in a gadget, while those in red cannot.

4.3 Problem Formulation

The goal of this section is to precisely state our synthesis problem, which operate on our intermediate representation (IR). To set the stage, we first present the syntax of

$\mathcal{L}_{\text{ZK}}$, a typical high-level DSL for ZKP (Section 4.3.1). We then give the syntax and the semantics of our novel IR, $\mathcal{L}_{\text{IR}}$ (Section 4.3.2), and a set of translation rules from $\mathcal{L}_{\text{ZK}}$ to $\mathcal{L}_{\text{IR}}$. We finally give a precise definition of the synthesis problem in Section 4.3.4.

4.3.1 The High-Level ZK DSL $\mathcal{L}_{\text{ZK}}$

The high-level source language that we consider here — which we shall denote by $\mathcal{L}_{\text{ZK}}$ — is extended from a standard imperative language with (a fixed) finite field $\mathbb{F}$ as the base type and basic arrays, similar to popular ZK DSLs such as [36, 22, 53]. As shown in Figure 4.4 (a), the DSL is expressive, supporting arithmetic computation and basic control flow such as loops and conditional. The DSL comprises two main constructs: expressions and statements. An expression can be a field constant $f \in \mathbb{F}$, a variable, a binary field operation, or an array read. Supported binary operations are addition $+$, subtraction $-$, and multiplication $\times$, which are the basic operations needed for field arithmetic.

Statements represent program logic and actions, such as variable assignments, array writes, sequencing, loops, conditionals, and assertions. An assignment writes the value of an expression to a mutable variable, while array writes update specific elements in arrays. A REPEAT loop executes its body statement for certain number of iterations, and conditionals execute different branches depending on a boolean expression. Assertions are made via `constrain  $e_1 = e_2$` , which checks if two expressions evaluate to the same field element.

This DSL is particularly suited for defining arithmetic computations, program flow, and constraint systems in a structured manner. Its integration of loops, conditionals, and assertions makes it a powerful tool for applications in formal verification, cryptographic protocols, and privacy-preserving systems, enabling the systematic construction and validation of complex computations.

$$\begin{array}{c}
\frac{}{\iota, \nu \vdash f \Downarrow f} \text{G-CONST} \qquad \frac{x = \sigma(\iota + \delta)}{\iota, \nu \vdash \sigma[i + \delta] \Downarrow \nu(x)} \text{G-ROT} \\
\\
\frac{\iota, \nu \vdash e_1 \Downarrow f_1 \quad \iota, \nu \vdash e_2 \Downarrow f_2 \quad f_3 = f_1 \oplus f_2}{\iota, \nu \vdash e_1 \oplus e_2 \Downarrow f_3} \text{G-BINOP} \\
\\
\frac{\iota, \nu \vdash e \Downarrow f \quad x = \sigma(\iota + \delta)}{\iota, \nu \vdash (\text{write } \sigma[i + \delta] \leftarrow e) \longrightarrow \nu[x \mapsto f]} \text{G-ASSIGN} \\
\\
\frac{\iota, \nu \vdash e_1 \Downarrow f_1 \quad \iota, \nu \vdash e_2 \Downarrow f_2 \quad f_1 = f_2}{\iota, \nu \vdash (\text{constrain } e_1 = e_2) \longrightarrow \nu} \text{G-CONSTRAIN} \qquad \frac{}{\iota, \nu \vdash (\cdot) \longrightarrow \nu} \text{G-STEP0} \\
\\
\frac{\iota, \nu \vdash s \longrightarrow \nu' \quad \iota, \nu' \vdash s' \longrightarrow \nu''}{\iota, \nu \vdash (s; s') \longrightarrow \nu''} \text{G-STEPS} \qquad \frac{}{\iota, \nu \vdash G\langle 0 \rangle (\overrightarrow{S : \sigma}) \longrightarrow \nu} \text{G-UNROLL0} \\
\\
\frac{\begin{array}{c} n > 0 \quad G = \text{gadget } G \{ _ ; s \} \\ \iota, \nu \vdash [\overrightarrow{\sigma/S}]s \longrightarrow \nu' \quad \iota + 1, \nu' \vdash G\langle n-1 \rangle (\overrightarrow{S = \sigma}) \longrightarrow \nu'' \end{array}}{\iota, \nu \vdash G\langle n \rangle (\overrightarrow{S = \sigma}) \longrightarrow \nu''} \text{G-UNROLLS}
\end{array}$$

Figure 4.5: Semantics of gadgets.

However, because the original high-level language $\mathcal{L}_{\text{ZK}}$ contains complex control flows and language features that are hard for program synthesis and optimizations, we next introduce our circuit intermediate representation and the refined problem of synthesizing equivalent IR programs.

4.3.2 Circuit Intermediate Representation $\mathcal{L}_{\text{IR}}$

Our circuit IR $\mathcal{L}_{\text{IR}}$ provides a structured and formal way to represent the low-level computations and constraints of a circuit. The most important construct in $\mathcal{L}_{\text{IR}}$ is the gadget, which is designed to both capture the powerful *custom gate* primitive provided by PLONKish proof systems [12] and distill computational patterns common to ZK circuits, such as mapping (applying a transformation to an array of field values) and aggregation

(accumulating values across an array of field values.¹

The syntax of $\mathcal{L}_{\text{IR}}$ is shown in Figure 4.4 (b). $\mathcal{L}_{\text{IR}}$ removes the high-level constructs such as loops, conditionals, and arrays from $\mathcal{L}_{\text{ZK}}$, and supports a more restricted set of operations and constraints. The most important feature of $\mathcal{L}_{\text{IR}}$ is the *gadget* abstraction:

- A gadget is a special function that represents a reusable computational building block of circuits that is applied multiple times at consecutive time steps. A gadget is defined by a list of parameters, which we call *signals*. Each signal encodes a finite time-series of values.
- The body of a gadget is a sequence of *steps* define the gadget’s behavior at each time step. A step is a single action that can be either assignment or constraint. An assignment has the form `write  $S[i + \delta] \leftarrow e$` , where $S[i + \delta]$ refers to the δ -th element of signal S relative to the current time step (denoted by symbol i), and e is the expression whose value will be assigned to $S[i + \delta]$. A constraint has the form `constrain  $e_1 = e_2$` , which enforces that the two expressions evaluate to the same value.
- Expressions represent the values or computations within a step. They can be field constants, binary operations, and formal parameters. An expression can refer to the δ -th element of signal S relative to the current time (denoted by symbol i) step using rotation $S[i + \delta]$.
- Finally, a gadget can be expanded in an IR program with a set of signal literals (which will replace the formal signal parameters during execution), and a natural number n that specifies the number of times the gadget is applied. A signal literal σ is a sequence of IR variables $[x_{-n}, \dots, x_{-1}; x_0, \dots, x_{m-1}]$. The first part $[x_{-n}, \dots, x_{-1}]$

¹A primary example is the inner product computation, which maps two input arrays into one by multiplying them element-wise, and aggregating over the multiplied values using sum.

$$\begin{array}{c}
\frac{}{\nu \vdash \cdot \longrightarrow \nu} \text{M-EMPTY} \qquad \frac{\nu \vdash m \longrightarrow \nu' \quad 0, \nu' \vdash s \longrightarrow \nu''}{\nu \vdash m; s \longrightarrow \nu''} \text{M-STMT} \\
\\
\frac{\nu \vdash m \longrightarrow \nu' \quad 0, \nu' \vdash G\langle n \rangle(\overrightarrow{S = \sigma}) \longrightarrow \nu''}{\nu \vdash m; G\langle n \rangle(\overrightarrow{S = \sigma}) \longrightarrow \nu''} \text{M-GADGET} \\
\\
\frac{\nu \vdash m \longrightarrow \nu_0 \quad \nu_{i-1} \vdash s_{1i} \longrightarrow \nu_i \text{ for } 1 \leq i \leq n \quad \nu_n \vdash (s_{21}; \dots; s_{2n}) \bullet \dots \bullet (s_{m1}; \dots; s_{mn}) \longrightarrow \nu'}{\nu \vdash m; (s_{11}; \dots; s_{1n}) \bullet (s_{21}; \dots; s_{2n}) \bullet \dots \bullet (s_{m1}; \dots; s_{mn}) \longrightarrow \nu'} \text{M-TRACE}
\end{array}$$

Figure 4.6: Semantics of $\mathcal{L}_{\text{IR}}$.

is the initial state of the signal, and the second part $[x_0, \dots, x_{m-1}]$ is the main part of the signal. If σ is a signal literal, we use the notation $\sigma[0]$ to refer to the first non-initial element of σ .

The design of the gadget abstraction can also be thought of as an generalization of higher-order functions **map** and **fold** to support 1) simultaneous mapping of multiple arrays and 2) folding across multiple time steps, both of which are commonly exploited in PlonK circuits.

The semantics of gadget evaluation is given in Figure 4.5.² We use ι to denote the index of the current time step (initialized to 0), and ν to denote the current environment that maps variables to their (field) values (initialized to the environment of the gadget caller). The expression evaluation judgment $\iota, \nu \vdash e \Downarrow f$ evaluates a gadget expression to field element f , and the statement evaluation judgment $\iota, \nu \vdash s \longrightarrow \nu'$ executes statement s , and returns an updated environment ν' . In the rules, we assume that signal names have been substituted with their corresponding signal literals. Finally, we use the judgment $\nu \vdash p \longrightarrow \nu'$ to denote the evaluation of $\mathcal{L}_{\text{IR}}$ program p from environment ν to ν' , which sequentially evaluates each statement, loop trace, or gadget expansion in p . The

²We omit the semantics rules for non-gadget part of the IR, since they are standard.

```

1  gadget Sum {
2    signal A, B
3    B[i] <- B[i-1] + A[i]
4  }
5  main (x0, x1, x2, x3, expected) {
6    sum0 <- 0;
7    Sum<4>(
8      A = [x0, x1, x2, x3],
9      B = [sum0; sum1, sum2, sum3, sum4]
10   );
11   sum4 == expected
12 }

```

Figure 4.7: An IR program that computes the sum of four input variables using a gadget.

rules defining this judgment is straightforward, and can be found in Figure 4.6. Since a $\mathcal{L}_{\text{IR}}$ program p is a list of gadget definitions followed by a main body m , we define the judgment $\nu \vdash m \longrightarrow \nu'$, which informally means evaluating the main body m with initial environment ν does not violate any constraint and yields an output environment ν' . The rules defining this judgment evaluate each statement, gadget expansion, and loop trace in m in sequence by threading through an environment.

Example 4.1. Figure 4.7 shows an example program that contains a gadget to compute the sum of signal A and stores the intermediate results in signal B . The gadget uses $B[i]$ and $B[i - 1]$ to refer to the current element and the previous element of signal B , respectively. The main program calls this gadget with signal A containing a list of input variables, and signal B whose initial element is `sum_0`, followed by intermediate sum variables. The main program then checks if the final sum is equal to an expected input value.

4.3.3 From High-Level Program $\mathcal{L}_{\text{ZK}}$ to Circuit IR $\mathcal{L}_{\text{IR}}$

The translation process for converting a high-level $\mathcal{L}_{\text{ZK}}$ program into a gadget-free circuit IR is guided by a set of standard rules shown in Figure 4.8 [27]. These rules transform high-level constructs such as variable assignments, constraints, conditionals,

$$\begin{aligned}
h_1 \Delta h_2 &= \{x \mapsto l \mid x \mapsto l_1 \in h_1, x \mapsto l_2 \in h_2, l_1 \neq l_2, l \text{ fresh}\} \\
h_1 \sqcup h_2(x) &= \begin{cases} l & \text{if } x \mapsto l \in h_1 \Delta h_2 \\ h_1(x) & \text{otherwise} \end{cases} \\
\phi(h_1, h_2) &= [\text{write } l \leftarrow l_1 + l_2 \mid x \mapsto l \in h_1 \Delta h_2] \quad \frac{}{h, x \vdash h(x) \Rightarrow} \text{TR-READ} \\
&\frac{h(x) = v \quad l \text{ fresh}}{\gamma, h, (x := e) \vdash h[x \mapsto l] \dashv (\text{write } l \leftarrow \gamma \times v) \rightsquigarrow} \text{TR-WRITE} \\
&\frac{h, e_1 \vdash v_1 \Rightarrow \quad h, e_2 \vdash v_2 \Rightarrow}{\gamma, h, (\text{constrain } e_1 = e_2) \vdash h' \dashv (\text{constrain } \gamma \times (v_1 - v_2) = 0) \rightsquigarrow} \text{TR-CONS} \\
&\frac{h, e_1 \vdash 0 \Rightarrow \quad \gamma, h, c_1 \vdash h' \dashv p \rightsquigarrow}{\gamma, h, \text{if } e_1 \text{ then } c_1 \text{ else } c_2 \text{ join} \vdash h' \dashv p \rightsquigarrow} \text{TR-IFTRUE} \\
&\frac{h, e_1 \vdash f \Rightarrow \neq 0 \quad \gamma, h, c_2 \vdash h' \dashv p \rightsquigarrow}{\gamma, h, \text{if } e_1 \text{ then } c_1 \text{ else } c_2 \text{ join} \vdash h' \dashv p \rightsquigarrow} \text{TR-IFFALSE} \\
&\frac{h, e_1 \vdash v \Rightarrow \quad \gamma \times v, h, c_1 \vdash h_1 \dashv p_1 \rightsquigarrow \quad (1 - \gamma) \times v, h, c_2 \vdash h_2 \dashv p_2 \rightsquigarrow \quad h' = h_1 \sqcup h_2 \quad p' = \phi(h_1, h_2)}{\gamma, h, \text{if } e_1 \text{ then } c_1 \text{ else } c_2 \text{ join} \vdash h' \dashv (p_1; p_2; p') \rightsquigarrow} \text{TR-IF} \\
&\frac{\gamma, h, c_1 \vdash h' \dashv p_1 \rightsquigarrow \quad \gamma, h', c_2 \vdash h'' \dashv p_2 \rightsquigarrow}{\gamma, h, (c_1; c_2) \vdash h'' \dashv (p_1; p_2) \rightsquigarrow} \text{TR-SEQ} \\
&\frac{h, e \vdash 0 \Rightarrow}{\gamma, h, \text{repeat } e \{c\} \vdash h \dashv \epsilon \rightsquigarrow} \text{TR-REPEATSKIP} \\
&\frac{h, e \vdash f \Rightarrow \neq 0 \quad \gamma, h, c \vdash h' \dashv p \rightsquigarrow \quad \gamma, h', \text{repeat } (f - 1) \{c\} \vdash h'' \dashv p' \rightsquigarrow}{\gamma, h, \text{repeat } e \{c\} \vdash h'' \dashv (p \bullet p') \rightsquigarrow} \text{TR-REPEATSTEP}
\end{aligned}$$

Figure 4.8: Selected translation rules from $\mathcal{L}_{\text{ZK}}$ to $\mathcal{L}_{\text{IR}}$.

and loops into semantically equivalent lower-level representations. The translation uses standard techniques of partial evaluation (to unroll loops) and SSA transform (to represent

mutable variables as logical variables). We use heap μ to map $\mathcal{L}_{\text{ZK}}$ variables to $\mathcal{L}_{\text{IR}}$ symbolic locations, and field expression γ to represent the current path condition. We use the judgment $\gamma, \mu, s \vdash \mu' \dashv p \rightsquigarrow$ to denote that under path condition γ and heap μ , statement s is translated to p and updates the heap to μ' . The translation produces intermediate representations expressed in terms of assignments, constraints, and *loop traces*, which mark parts of the IR program as unrolled loops and are later used to synthesize gadgets. Moreover, since we use a field expression to represent the path condition γ , the TR-IF rule traverses the then-branch of an if-statement with a condition value v by updating the path condition γ to $\gamma \times v$, since boolean conjunction is equivalent to field multiplication.

Basic statements. Assignments are translated into IR write operations. A fresh symbolic location is allocated for the assigned value. Constraints are transformed into corresponding IR equality constraints after evaluating both expressions. Both write operations and constraints are guarded by the current path condition γ .

Conditionals. If the branch condition evaluates concretely to a constant, then the appropriate branch is taken. Otherwise, both the true branch and the false branch are explored under an augmented path condition. The resulting heaps are merged using the $\sqcup$ operator, which combines the two heaps by mapping variables with conflicting locations to fresh locations. Two sub-programs p_1 and p_2 are concatenated, and appended with $p' = \phi(\mu_1, \mu_2)$, which contains IR statements that represent the ϕ -nodes of the two branches.

Loop Translation. Loops with zero iterations are translated into an empty trace. For loops with multiple iterations, each iteration is translated recursively, and the resulting traces are concatenated with $\bullet$.

4.3.4 Problem Definition

In this section, we formally state the main synthesis problem. Before doing so, we define several auxiliary definitions.

Definition 4.2. (Partial program) Given a $\mathcal{L}_{\text{IR}}$ program p with loop traces, a *partial program* p' is obtained from p by replacing each loop trace with a hole.

The goal of synthesis is to find the best *completion* of a partial program by filling in each hole with a gadget expansion expression g (or a gadget g for short). To quantify the notion of cost between different gadgets, we define the following generic cost function over gadgets:

Definition 4.3. (Gadget cost function Θ) The gadget cost function $\Theta : g \rightarrow \mathbb{N}$ maps a given gadget g to its approximate cost in natural number.

The cost function Θ can be instantiated to take into account multiple factors that would impact the performance of the prover (i.e., proving time and proof size), such as the number of signals declared or the depth of g .

While Θ provides an abstract measure of a gadget's cost to guide synthesis at the $\mathcal{L}_{\text{IR}}$ level, the *actual* cost of a gadget cannot be measured until it has been fully compiled into a circuit. This is because each gadget $g \in G$ is ultimately realized as a set of constraints embedded into a tabular layout, with its cost reflected by the number of distinct polynomial gates it uses, along with the number of rows in which those gates are active. Formally, the layout cost of a gadget can be expressed as:

$$\varsigma(g) = |G_{\text{enabled}}| \cdot |M(g)|,$$

where G_{enabled} is the set of polynomial gate expressions generated by g , and $M(g)$ is the set of rows over which those gates are instantiated. The goal is to realize each gadget g

in a way that minimizes the cost:

$$g_{\diamond} = \arg \min_g \varsigma(g).$$

Although ς operates at a lower level of abstraction than Θ , both aim to quantify implementation cost, and are closely aligned in capturing resource usage that impacts prover performance.

Since we wish to synthesize $\mathcal{L}_{\text{IR}}$ programs that are equivalent to some specification $\mathcal{L}_{\text{IR}}$ program, we now define the notion of equivalence. Intuitively, two $\mathcal{L}_{\text{IR}}$ programs p_1 and p_2 are equivalent if and only if from some initial environment, evaluating p_1 and p_2 either both succeed (passing all constraints), or both fail (violating at least one constraint). The precise definition is given below.

Definition 4.4. ($\mathcal{L}_{\text{IR}}$ equivalence) Given two $\mathcal{L}_{\text{IR}}$ programs p_1 and p_2 , we say p_1 is *equivalent* to p_2 if for any environment ν , there exists some ν_1 such that $\nu \vdash p_1 \longrightarrow \nu_1$ if and only if there exists some ν_2 such that $\nu \vdash p_2 \longrightarrow \nu_2$. We use the notation $p_1 \equiv p_2$ to denote that p_1 is equivalent to p_2 .

Definition 4.5. (Program synthesis) Given a reference program p in $\mathcal{L}_{\text{IR}}$ and a partial program p' obtained from p , the goal of program synthesis is to find a *complete* program $p_{\diamond}$ by filling the holes with gadgets such that a) $p_{\diamond} \equiv p$ according to the semantics of $\mathcal{L}_{\text{IR}}$, and b) the cost of $\Theta(p)$ is minimized.

4.4 Circuit Synthesis Guided by Abstract Semantics

In this section, we first overview our synthesis algorithm. We will then explain the salient components of the algorithm in detail.

Algorithm 3: Synthesis algorithm.

Input: Initial IR program p lowered from source program in $\mathcal{L}_{\text{ZK}}$
Output: Optimized IR program $p_\diamond$

```

1  $C, p' \leftarrow \text{DECOMPOSE}(p);$  // decompose  $p$  into components  $C$  and partial
   program  $p'$ 
2  $\mathcal{G} \leftarrow \emptyset;$  // initialize solutions
3  $\mathcal{E} \leftarrow \text{GENRANDOMEXAMPLES}();$  // initialize examples
4 for each  $c \in C$  do
5    $g^* \leftarrow \text{INFERSKETCH}(c);$  // infer a symbolic gadget sketch
6   while  $g \leftarrow \text{ENUMERATE}(g^*)$  do
7     if  $\text{EQUIV}(g, c)$  then
8        $\mathcal{G}[c] \leftarrow g;$  // add synthesized gadget  $g$  to solutions
9       break;
10    else
11       $\mathcal{E} \leftarrow \text{CEX}(g, c);$  // learn counterexample
12    end
13  end
14 end
15  $p_\diamond \leftarrow \text{ASSEMBLE}(\mathcal{G}, p');$  // assemble synthesized  $\mathcal{G}$  and partial program
    $p'$ 
16 return  $p_\diamond;$ 

```

4.4.1 Overview

The synthesis algorithm is shown in Algorithm 3. At a high level, our synthesis procedure follows the well-established Counterexample-Guided Inductive Synthesis (CEGIS) paradigm [48, 50]. In CEGIS, the synthesizer maintains two complementary phases:

- a synthesis phase, which searches for a candidate program that satisfies a finite set of input–output examples, and
- a verification phase, which checks the candidate against the full specification and, if it fails, produces a counterexample to refine the search space.

Our algorithm instantiates this loop at the level of ZK circuit IRs: components extracted from the original program are synthesized into candidate gadgets, verified for equivalence

via concrete testing and SMT-based checking, and iteratively refined using counterexamples. This connection to CEGIS ensures that the overall flow in Algorithm 3 is principled and systematically explores the search space. We next outline how this process is tailored to our setting.

The algorithm takes as input an unoptimized IR program as input, and returns an optimized IR program $p_\diamond$ as output. Next, the synthesis algorithm decomposes the trace into a topologically sorted list C of gadget components using data-dependency information (Section 4.4.2) and a partial program p' in line 4. It then initializes the map $\mathcal{G}$ which maps specification components to synthesized gadgets to be empty. It also randomly generates a set of input examples.

The outer loop (line 7) iterates over each component in their sorted order. For each component $c \in C$, the algorithm performs a lightweight static analysis using INFERSKETCH (Section 4.4.3) to obtain a symbolic gadget sketch g^* that encodes the pruned search space.

In the inner loop (line 9), each feasible concrete gadget is enumerated from the sketch g^* in increasing syntactic complexity. Each candidate g is verified against the specification component c using the EQUIV procedure, which checks the equivalence between candidate g and the specification c using both concrete examples from $\mathcal{E}$ and an SMT-based bounded model checker (Section 4.4.4). If gadget g is determined to be equivalent to component c , the solution map $\mathcal{G}$ is updated to map c to solution g . Otherwise, the algorithm learns a counterexample from the SMT solver and visits the next candidate. The outer loop continues until all components have been synthesized. Finally, the procedure ASSEMBLE assembles the solutions $\mathcal{G}$, and composes them with the partial program p' to obtain an optimized IR program $p_\diamond$.

4.4.2 Decomposition

The purpose of the DECOMPOSE procedure is twofold. First, it identifies the parts of the IR program p that benefit the most from being represented as gadgets, which are typically recurring computation patterns. Thus, we locate and extract from the input IR program p the trace t_i for the i -th innermost loop:

$$t_i = (s_{11}; \dots; s_{1n}) \bullet (s_{21}; \dots; s_{2n}) \bullet \dots \bullet (s_{m1}; \dots; s_{mn}),$$

where m is the number of iterations in the loop trace t_i , and n is the number of statements in each iteration³. We refer to each $(s_{j1}; \dots; s_{jn})$ as the j -th *fragment* of the loop trace t_i .

Once all innermost loop traces have been extracted from the input IR program p , we replace them with *holes* that will be filled with gadgets yet to be synthesized, resulting in a partial program p' . Thus, an IR program p is decomposed into a set of innermost loop traces $\{t_1, \dots, t_k\}$ and a partial program p' . Once synthesis is complete, the i -th hole of the partial program p' will be replaced with the synthesized gadget(s) for trace t_i .

The second purpose of decomposition is to make synthesis scalable by further breaking down each synthesis problem t_i into a sequence of *components* $C = [c_1, c_2, \dots, c_L]$, such that each component $c_l \in C$ is a strictly smaller slice of the original loop trace t_i , and the composition of the solutions to components $c_1, \dots, c_L$ gives a solution to the original problem t_i . To obtain such a component-based decomposition for a trace t_i with m fragments and n statements per fragment, we construct a data-flow graph $G = (V, E)$, where $V = \{1, \dots, n\}$ represents the indices of statements relative to the beginning of the fragment. In graph G , there is an edge $(k_1, k_2) \in E$ if and only if there is a write to some variable x in the k_1 -th statement and a read from the same variable x in the k_2 -th statement in some fragment. We label each edge with the set of variables that are read

³ n must be the same for all iterations of the same loop, since ZK programs lack genuine control-flow

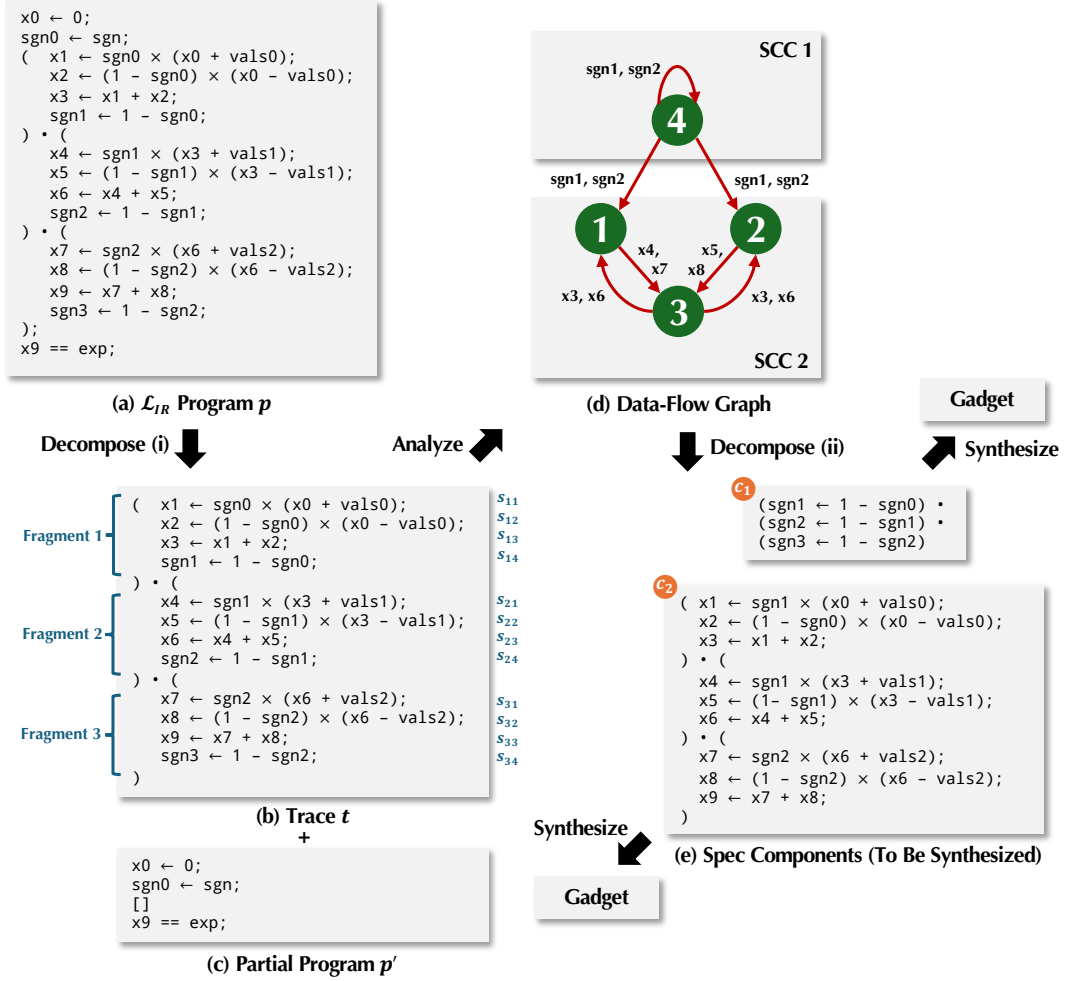


Figure 4.9: Illustration of an example decomposition procedure.

and written along the edge. Finally, we compute the *strongly connected components* of G to obtain the set of synthesis components, and topologically sort them to obtain the sequence C of components.

Example 4.6. Consider the $\mathcal{L}_{IR}$ trace program p shown in Figure 4.9, which is generated from the $\mathcal{L}_{ZK}$ program Figure 4.3. The program p has a single innermost loop trace t , with 3 fragments. Decomposition first breaks down p into trace t and the partial program p' (b). It then constructs the dataflow graph shown in (c). Each node i in the graph represents the i -th statement in each fragment. There is an edge from node 4 to node 1,

since variables `sgn0` and `sgn1` are written in statements s_{14} and s_{24} , and read in s_{21} and s_{31} . The strongly-connected components consist of $\{4\}$ and $\{1, 2, 3\}$, which give rise to components R and S to be synthesized.

4.4.3 Sketch Inference

Given the set C of decomposed components, we are ready to synthesize gadgets for each component. Each gadget operates on a set of *signals*, each of which is a sequence of variables that occur in the loop trace. We let S be the set of signals that can be extracted from the $\mathcal{L}_{\text{IR}}$ trace program p by identifying the elements that occur in the same relative position in each fragment as belonging to the same signal. Two signals s_1 and s_2 are coalesced if a postfix of s_1 overlaps with a prefix of s_2 . We use the notation $\sigma^{-1}(x) = s$ to denote that we identify element x as belonging to signal s ⁴.

Since each component $c \in C$ may still be complex, we first infer a *sketch*, which is a feasible but incomplete gadget program for c . The inference procedure `INFERSKETCH` relies on a static analysis to compute the following two sets of signals from the loop trace:

- R : an over-approximating set of signals that can be read in the gadget sketch.
- W : an over-approximating set of signals that can be written in the gadget sketch.

Definition 4.7. (Abstract semantics over read-write set) Given a program p in language $\mathcal{L}_{\text{IR}}$, we introduce an abstraction relation $p \Rightarrow_{\xi} S$ where $\xi \in \{R, W\}$ to compute its read-write set of signals using the inference rules in Figure 4.10.

Each rule in Figure 4.10 defines the read-write set semantics for the corresponding construct in $\mathcal{L}_{\text{IR}}$. These semantics are used to compute the over-approximation set of signals that a program reads from (R) or writes to (W). Then, when enumerating

⁴Every occurrence of a variable can only belong to exactly one signal.

$$\begin{array}{c}
\frac{}{f \Rightarrow_R \emptyset} \text{READ-CONST} \qquad \frac{\sigma^{-1}(x) = s}{x \Rightarrow_R \{s\}} \text{READ-VAR} \\
\\
\frac{e_1 \Rightarrow_R S_1 \quad e_2 \Rightarrow_R S_2}{e_1 \oplus e_2 \Rightarrow_R S_1 \cup S_2} \text{READ-BINOP} \qquad \frac{e \Rightarrow_R S}{(\text{write } x \leftarrow e) \Rightarrow_R S} \text{READ-ASSIGN} \\
\\
\frac{e_1 \Rightarrow_R S_1 \quad e_2 \Rightarrow_R S_2}{(\text{constrain } e_1 = e_2) \Rightarrow_R S_1 \cup S_2} \text{READ-CONSTRAIN} \\
\\
\frac{\sigma^{-1}(x) = s}{(\text{write } x \leftarrow e) \Rightarrow_W \{s\}} \text{WRITE-ASSIGN} \\
\\
\frac{}{(\text{constrain } e_1 = e_2) \Rightarrow_W \emptyset} \text{WRITE-CONSTRAIN} \\
\\
\frac{}{\cdot \Rightarrow_\xi \emptyset} \text{STEP0} \qquad \frac{s_1 \Rightarrow_\xi S_1 \quad s_2 \Rightarrow_\xi S_2}{(s_1; s_2) \Rightarrow_\xi S_1 \cup S_2} \text{STEPS}
\end{array}$$

Figure 4.10: Sketch inference rules.

candidate gadgets, we only explore candidates g whose abstract semantics are consistent with the abstracted $\hat{p}$.

Example 4.8. Consider the specification component c_2 from Figure 4.9. Suppose we identify from the loop trace the signal $s = [\text{vals0}, \text{vals1}, \text{vals2}]$. Since the variables in signal s are read in c_2 , if $c_2 \Rightarrow_R S$, then $s \in S$ by the READ-VAR rule.

4.4.4 Equivalence Checking

Once a gadget g is enumerated by Algorithm 3, a procedure $\text{EQUIV}(g, c)$ is then invoked to verify the equivalence between the proposed gadget g and the specification component c . Given the difficulty of scaling a fully-fledged symbolic equivalence checker to real-world circuits, we design EQUIV as a hybrid procedure that consists of two phases to improve its scalability: a testing phase that incorporates test cases to quickly prune

infeasible gadgets using concrete semantics, and a verification phase that incorporates a bounded model checker to reason about potential violation of the gadget against the component.

Test case generation. To verify the correctness of gadgets, the synthesizer runs a candidate gadget against the specification loop trace on a set of concrete input test cases. We use a combination of randomly generated inputs and actual witness values for these test cases. These witness values are obtained by concretely evaluating the IR program from prover-provided input values in a standard way ([22]).

Bounded model checking. If a candidate that passes all concrete test cases, the EQUIV procedure then symbolically evaluates both the gadget g and the specification component c on a set of symbolic inputs and checks whether their resulting output states are equivalent. The symbolic evaluator is directly adapted from the concrete semantics as described in Figure 4.5.

The SMT formula is built as follows. Let V_{RO} be the set of variables that are read but not written to in a specification component c , and V_W be the set of variables that are written to in c . Then, two $\mathcal{L}_{\text{IR}}$ programs p_1 and p_2 are equivalent if and only if all variables $x \in V_W$ are assigned the same values, and that the conjunction of all constraints in p_1 holds if and only if the conjunction of all constraints in p_2 holds. The resulting SMT formula can be directly discharged by off-the-shelf solvers such as CVC5 [95] equipped with the theory of finite fields [42].

4.5 Implementation

In this section, we provide more details about the implementation of TABBY, which is implemented in approximately 7000 lines of OCaml. TABBY supports generating

PLONKish circuits to the Halo2 backend. Given an optimized IR program, TABBY first translates it into a PLONKish circuit in a syntax-directed way. TABBY also outputs a witness computation program to allow the prover to supply input variables, from which the values of the remaining table cells can be automatically derived, as standard [22].

PLONKish circuit generation. Since an optimized IR program consists of a gadget-free part and one or more gadgets, we describe their translation separately. Each assignment in the gadget-free part is transformed into a 3-address code that contains addition and multiplication of variables, or variable/constant assignments. Constraint statements are normalized into the form $x = 0$, where x is a variable. After this translation, all assignments and constraints can be represented using PLONKish standard gates and copy constraints (which copy operands to adjacent locations). For each gadget, every step is translated into a custom gate in a straightforward fashion. We only use a single table column to simplify the layout. So all signals in a gadget are interleaved into the same column. All steps in the same gadget share the same fixed selector column. The final circuit is the concatenation of the gadget-free part and all gadgets. We pass the resulting system of polynomial equations and the concrete table values are passed to the Halo2 backend via PLAF to generate the final circuit and proof.

4.6 Evaluation

In this section, we present the design and results of the evaluation for answering the following research questions:

- **RQ1 (Circuit Performance):** How do the circuits generated by TABBY perform compared to those generated by state-of-the-art ZK compilers?

- **RQ2 (Synthesis Performance)**: How effective does TABBY perform for synthesizing optimized circuits? How significant is the benefit of each key design in TABBY’s core algorithm?
- **RQ3 (Generality)**: How effective is TABBY’s algorithm in optimizing circuits in other ZK programming languages and proof systems?

Benchmarks We collected a set of 23 benchmarks from 8 real-world open-source projects written in various ZK programming languages and zero-knowledge virtual machine (zkVM) (e.g., Noir [36], SP1 [29] and Circom [22]), which contains wide coverage of usage of different ZK language constructs for various computational tasks and libraries, e.g., math operations, games, cryptographic utilities, etc. Table 4.1 shows the basic statistics of the benchmark suite: overall, each benchmark has an average of 5986 LOC after the translation to $\mathcal{L}_{\text{IR}}$.

Evaluation metrics To quantitatively evaluate TABBY, we organize our evaluation into two phases. The first phase focuses on the performance of the synthesized circuits (RQ1), where the effectiveness of each circuit is evaluated via the following metrics:

- **Compile time** measures the time spent compiling the high-level program into circuit.
- **Proof size** measures the length of the proof generated for a circuit by the backend cryptographic library.
- **Proving time** measures the time spent for a prover to generate a proof with the target circuit.
- **Table size** measures the total number of cells of the assigned table, including the input/witness/fixed cells.

Table 4.1: Basic statistics of the benchmarks collected. Note that “Utils” is a collection of multiple projects.

ZK Language	Project	Benchmark	LOC
Noir	BattleZips	has_ship	8,197
		check_for_collision	9,625
		check_ship_range	630
		check_for_hit	1,191
	Sudoku	check_cards	169
	Utils	dotproduct	12,292
		fib	105
		pow	259
		sqrt	2,098
SP1	Utils	sort	205
Circom	Utils	Num2Bits	4,104
		MultiMux1	12,292
		Decoder	127
		MultiAND	8,197
	BigInt	BigIsEqual	3,014
		BigLessThan	18,196
		BigAddNoCarry	9,192
		BigSub	17,868
		BigAdd	9,258
		BigModSumThree	2,067
		BigModSumFour	2,069
		ModProd	518
		BigSubModP	27,190
Average			6,472

For the second phase (RQ2), we evaluate the performance of the synthesis algorithm from TABBY itself along with its core design choices, using the *synthesis time* as a metric to measure the end-to-end time cost for TABBY to consume the original circuit and produce the optimized one.

Experiment setup All experiments reported in Section 4.6.1 and Section 4.6.2 are conducted on a MacBook Pro with an Apple M1 Pro CPU and 16 GB of memory. TABBY is written in OCaml and compiled with OCaml 5.1.1. We evaluate the PLONKish circuits

generated by TABBY against those generated by Noir (version 0.8.0), CirC (branch zsharp), and Circom (version 2.2.1). All compilers are connected to a Halo2 backend of version 0.3.0 [96].⁵ The default timeout for evaluation of each benchmark is set to 60 seconds for proving and 10 minutes for synthesis. The experiment of the case study in Section 4.6.3 is conducted on a Linux machine equipped with 16 GB of memory, a 64-bit Intel(R) CPU @ 3.00 GHz with 8 cores, and a Ubuntu 22.04 operating system.

4.6.1 Performance of Synthesized Circuits (RQ1)

To find out how the synthesized circuits from TABBY perform, we compare them with those compiled from Noir [36], Circom [22] and CirC [27], the state-of-the-art ZK programming languages and compilers. We report compile and proving time in Table 4.2, Table 4.4 and Table 4.6; proof size and table size are reported in Table 4.3, Table 4.5 and Table 4.7.

Both TABBY and Noir/Circom/CirC solve all the benchmarks and successfully generate their corresponding circuits. Since Halo2 circuits have a tabular form, which in practice is a key factor that affects the performance of a circuit, we measure the size of circuit table for each benchmark in terms of the number of cells. As shown in the **Table Size** column of both tables, TABBY greatly reduces the table size, with a median of 17,742 (resp. 79,366 and 17,742) cells compared to a median of 55,080 (resp. 983,040 and 245,760) cells for circuits compiled by Noir (resp. Circom and CirC), yielding a *significant* reduction of 90.5% (resp. 87.4% and 96.5%) as the average and 67.8% (resp. 91.8% and 92.8%) as the median.

In terms of the proof size, circuits generated by Noir and Circom have the same size

⁵The latest version of the CirC compiler uses incompatible Rust features against Halo2; hence, we use an earlier version of CirC that supports Rust 1.82. For Circom-generated circuits, we employ a simple syntax-directed translation to convert the output in R1CS format to Halo2 circuits. For Circ-generated circuits, we integrate CirC as a library within Halo2 and convert its R1CS output to the unified R1CS format shared with Circom.

Table 4.2: Compile time and proving time: TABBYP vs. Noir.

Benchmark	Compile Time (s)		Proving Time (s)	
	Noir	TABBYP	Noir	TABBYP
fib	1.03	0.43	0.16	0.05 (-68.8%)
check_for_collision	37.58	25.82	29.0	0.48 (-98.3%)
check_ship_range	1.21	2.05	1.04	0.71 (-31.7%)
has_ship	37.10	11.91	28.51	0.66 (-97.7%)
check_hit	1.41	0.22	0.60	0.05 (-91.7%)
dotproduct	17.62	6.94	16.15	3.52 (-78.2%)
check_cards	1.70	1.38	29.30	0.23 (-99.2%)
pow	1.98	2.20	1.54	0.49 (-68.2%)
sqrt	3.27	2.84	2.91	0.12 (-95.9%)
sort	1.45	3.74	0.60	0.23 (-61.7%)
Averaged	10.43	5.75	10.98	0.65 (-94.0%)
Median	1.84	2.52	2.22	0.36 (-84.0%)

(i.e., 1760 bytes) since they always use the standard PLONK gate in their compilation, which produces a fixed number of columns for the circuit table and thus results in a constant proof size. Besides standard gates, TABBYP is capable of synthesizing custom gates on a per-circuit basis, which helps it reduce the proof size for an average of 516 bytes across all benchmarks, yielding a 70.9% (resp. 70.5% and 71.8%) size reduction compared to Noir (resp. Circom and CirC), with a maximum reduction of 78.2% in the `fib` benchmark, cutting the proof size from 1760 bytes down to 384 bytes. For the worst-case scenario, TABBYP is still able to reduce the proof size by more than 60%. In the comparison of the actual proving time for each compiled circuit, TABBYP reduces the prover time cost by 94.0% (resp. 91.0% and 92.7%) on average compared to that compiled from Noir (resp. Circom and CirC).

In terms of compile time, the results show that TABBYP consistently achieves comparable compilation latency compared to Noir, Circ, and Circom. As shown in the **Compile Time** column of all tables, TABBYP completes compilation in 5.74 (resp. 9.91, 6.56) seconds on average compared to 10.43 (resp. 1.36, 0.96) seconds for circuits compiled by

Table 4.3: Proof size and table size: TABBY vs. Noir.

Benchmark	Proof Size (bytes)		Table Size (# of cells)	
	Noir	TABBY	Noir	TABBY
fib	1,760	384 (-78.2%)	450	125 (-72.2%)
check_for_collision	1,760	608 (-65.4%)	1,247,220	223,660 (-82.1%)
check_ship_range	1,760	544 (-69.1%)	12,600	8,365 (-33.6%)
has_ship	1,760	512 (-70.9%)	1,474,740	102,440 (-93.1%)
check_hit	1,760	416 (-76.4%)	13,770	11,244 (-18.3%)
dotproduct	1,760	448 (-74.5%)	180,180	61,470 (-65.9%)
check_cards	1760	544 (-69.1%)	1,888,470	24,240 (-98.7%)
pow	1,760	512 (-70.9%)	31,950	5,115 (-84.0%)
sqrt	1,760	640 (-63.6%)	78,210	31,626 (-59.6%)
sort	1,760	512 (-70.9%)	9,180	2,540 (-72.3%)
Averaged	1,760	512 (-70.9%)	493,677	47,083 (-90.5%)
Median	1,760	512 (-70.9%)	55,080	17,742 (-67.8%)

Noir, Circom, and Circ. Across all benchmarks, the compilation of TABBY terminates within 30 seconds, with a maximum of 29.94 seconds on BigSubMoP and a minimum of 0.43 seconds on fib.

Result for RQ1: TABBY generates optimized circuits that have *significantly* improved performance. In addition, the optimization strategy of TABBY brings more *consistent* improvement across different kinds of circuits.

4.6.2 Performance of Synthesis Algorithm and Ablation Study (RQ2)

To find out how each of TABBY’s design choices affects the core algorithm, we conduct an experiment that compares the synthesis performance of the tool. In particular, we study the difference brought by *sketch inference* (Section 4.4.3) and *decomposition* (Section 4.4.2); to do this, we create the following ablative versions of TABBY:

Table 4.4: Compile time and proving time: TABBY vs. CirC.

Benchmark	Compile Time (s)		Proving Time (s)	
	CirC	TABBY	CirC	TABBY
fib	0.06	0.43	0.14	0.05 (-64.3%)
check_for_collision	2.29	25.82	52.06	5.95 (-88.6%)
check_ship_range	1.94	2.05	48.93	0.71 (-98.5%)
has_ship	2.70	11.91	46.27	0.66 (-98.6%)
check_hit	0.05	0.22	0.33	0.05 (-84.8%)
dotproduct	0.56	6.94	4.09	3.52 (-13.9%)
check_cards	0.06	1.38	4.13	0.23 (-94.4%)
sort	0.02	3.74	0.51	0.23 (-54.9%)
Averaged	0.96	6.56	19.56	1.43 (-92.7%)
Median	0.31	2.90	4.11	0.45 (-89.2%)

- **The Baseline Version (“Base.”)** This is TABBY without sketch inference or decomposition.
- **The Sketch Inference Version (“+S”)** This is the baseline with sketch inference only.
- **The Decomposition Version (“+D”)** This is the baseline with decomposition only.

Also, we show the results in Table 4.8, where we refer to the full-fledged version of TABBY as “Full” for short.

As it takes an average of 7.32 seconds for the full version to optimize a circuit, it is solving more benchmarks than all other ablative versions. In particular, it is over $54\times$ faster than the baseline version (respectively, $54\times$ than the +S version and $18\times$ than the +D version). A closer look at the benchmarks solved indicates that both the sketch inference and decomposition actually help scale the algorithm up to more complex benchmarks. Specifically, decomposition is critical in ensuring that synthesis can terminate within the 10-minute timeout period, while sketch inference helps further cut down the

Table 4.5: Proof size and table size: TABBYP vs. CirC.

Benchmark	Proof Size (bytes)		Table Size (# of cells)	
	CirC	TABBYP	CirC	TABBYP
fib	1,760	384 (-78.2%)	480	125 (-74.0%)
check_for_collision	1,760	608 (-65.5%)	3,932,160	223,660 (-94.3%)
check_ship_range	1,760	544 (-69.1%)	3,932,160	8,365 (-99.8%)
has_ship	1,760	512 (-70.9%)	3,932,160	102,440 (-97.4%)
check_hit	1,760	416 (-76.4%)	7,680	11,244 (+46.4%)
dotproduct	1,760	448 (-74.5%)	245,760	61,470 (-75.0%)
check_cards	1760	544 (-69.1%)	245,760	24,240 (-90.1%)
sort	1,760	512 (-70.9%)	15,360	2,540 (-83.5%)
Averaged	1,760	496 (-71.8%)	1,538,940	54,260 (-96.5%)
Median	1,760	512 (-70.9%)	245,760	17,742 (-92.8%)

synthesis time.

Result for RQ2: TABBYP performs more scalable synthesis than the baseline, and both of its core design choices (sketch inference and decomposition) are important to TABBYP’s synthesis performance.

4.6.3 A Case Study of TABBYP for Optimizing Other Languages (RQ3)

Although we target PLONKish circuits in the previous experiments, with additional engineering effort, TABBYP can be adapted to a similar arithmetization that supports a primitive notion of state-transition functions via custom gates. In this case study, we describe how TABBYP can be applied to synthesize optimized AIR [97] circuits, another popular arithmetization behind zkSTARKs [14]. In AIR, the circuit is also represented by a table. Each row of the table is a state, and constraints can be either state initialization, state transition functions that are applied to intermediate states, and state finalization.

Table 4.6: Compile time and proving time: TABBY vs. Circom.

Benchmark	Compile Time (s)		Proving Time (s)	
	Circom	TABBY	Circom	TABBY
Decoder	0.11	2.85	1.06	0.38 (-64.2%)
Num2bits	6.21	22.71	17.41	0.59 (-96.6%)
Multimux1	1.02	27.28	30.34	4.95 (-83.7%)
BigIsEqual	0.62	3.31	17.01	2.40 (-85.9%)
MultiAND	1.07	2.51	15.30	3.33 (-78.2%)
BigLessThan	1.84	7.63	7.91	0.29 (-96.3%)
BigAddNoCarry	1.73	8.12	22.84	1.03 (-95.5%)
BigSub	1.75	17.41	38.17	1.83 (-95.2%)
BigAdd	1.71	14.02	25.36	0.86 (-96.6%)
BigModSumThree	0.12	7.34	1.35	0.27 (-80.0%)
BigModSumFour	0.10	7.86	1.32	0.29 (-78.0%)
ModProd	0.11	0.44	2.22	0.03 (-98.6%)
BigSubModP	0.26	29.94	98.25	1.65 (-98.3%)
Averaged	1.36	10.12	15.02	1.35 (-91.0%)
Median	1.04	7.99	17.01	0.86 (-94.9%)

We use the Fibonacci computation as our case study. The source program in $\mathcal{L}_{\text{ZK}}$ is shown in Figure 4.11 (b). Given this program as input, TABBY finds an optimized IR program, in which the Fibonacci computation is summarized as the (obvious) custom gate that computes the current Fibonacci number based on the previous two numbers. The corresponding AIR program, written in an idealized AIR DSL (similar to Plonky3 [92]), is shown in Figure 4.11 (c). The corresponding AIR table filled with witness values is shown in Figure 4.11 (d).

Empirically, we find that TABBY can synthesize AIR circuits with a similar performance as the PLONKish circuits. When the loop of the Fibonacci program is executed 2^{20} times, we obtain an AIR table of size $2 * 2^{20}$, as expected. Proving this circuit in Plonky3⁶ results in a proving time of 30.5 seconds. For comparison, we compile a similar Fibonacci program using SP1 [29], a ZK compiler implemented in Plonky3 that supports Rust as

⁶using the default Mersenne31 field with prime $p = 2^{31} - 1$

Table 4.7: Proof size and table size: TABBY vs. Circom.

Benchmark	Proof Size (bytes)		Table Size (# of cells)	
	Circom	TABBY	Circom	TABBY
Decoder	1,760	544 (-69.1%)	30,720	3,918 (-87.2%)
Num2bits	1,760	480 (-72.7%)	983,040	81,970 (-91.7%)
Multimux1	1,760	416 (-76.4%)	1,966,080	81,935 (-95.8%)
BigIsEqual	1,760	576 (-67.3%)	983,040	30,126 (-96.9%)
MultiAND	1,760	416 (-76.4%)	491,520	40,985 (-91.7%)
BigLessThan	1,760	544 (-69.1%)	491,520	79,366 (-83.9%)
BigAddNoCarry	1,760	544 (-69.1%)	1,966,080	241,626 (-87.7%)
BigSub	1,760	544 (-69.1%)	1,966,080	473,298 (-75.9%)
BigAdd	1,760	544 (-69.1%)	1,966,080	241,605 (-87.7%)
BigModSumThree	1,760	544 (-69.1%)	61,440	57,568 (-6.30%)
BigModSumFour	1,760	544 (-69.1%)	61,440	57,582 (-6.28%)
ModProd	1,760	544 (-69.1%)	122,880	3,731 (-96.9%)
BigSubModP	1,760	576 (-67.3%)	7,864,320	819,568 (-89.6%)
Averaged	1,760	520 (-70.5%)	1,458,018	170,252 (-87.4%)
Median	1,760	544 (-69.1%)	983,040	79,366 (-91.8%)

the source language.⁷ We obtain a proving time of 1,501 seconds. That is, TABBY generates a circuit that yields a 50× speed up in proving time.

Result for RQ3: TABBY can be generalized to optimize circuits and achieve performance gains in other arithmetization schemes.

⁷We note that SP1 uses a VM-based architecture and leverages recursive proofs, so the running times may not be comparable.

Table 4.8: A comparison of synthesis performance between TABBY and its ablative versions. “-” means statistical data is not available due to timeout for the corresponding ablative version.

Benchmark	Synthesis Time (s)			
	Base.	+S	+D	Full
has_ship	-	-	-	12.51
check_for_collision	-	-	26.92	4.39
check_ship_range	-	-	1.55	0.11
check_hit	0.01	0.01	0.01	0.01
check_cards	0.11	0.10	0.11	0.10
dotproduct	6.02	6.02	6.20	6.04
fib	0.01	0.01	0.01	0.01
pow	-	-	1.98	1.93
sqrt	-	-	2.01	1.98
sort	-	-	7.82	7.17
Num2Bits	-	-	4.00	0.35
MultiMux1	0.93	0.95	0.93	0.93
Decoder	-	-	0.26	0.07
MultiAND	1.04	1.03	1.04	1.05
BigIsEqual	-	-	0.21	0.06
BigLessThan	-	-	8.68	0.82
BigAddNoCarry	-	-	-	11.47
BigSub	-	-	-	11.79
Average	>400	>400	>136	7.32

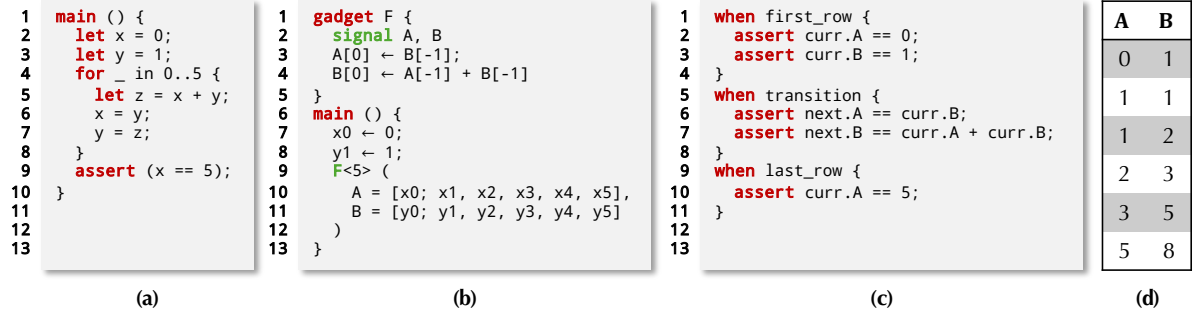


Figure 4.11: A case study of Fibonacci computation in AIR. (a) A $\mathcal{L}_{\text{ZK}}$ program that computes the 10-th Fibonacci number; (b) The Fibonacci gadget synthesized by TABBY; (c) The AIR constraints of Fibonacci computation; (d) The AIR table.

4.7 Related Work

In this section, we examine the most closely related works relevant to our proposed approach.

Zero-knowledge proof systems and arithmetizations zk-SNARK [61] is one of those prevailing ZK proof systems used in real-world privacy-aware application domains, such as blockchains and authentication systems, where anonymous transactions and trustless verification are indispensable. There are many realizations and variants of the zk-SNARK proof system, such as Groth16 [9] and PLONK [10], where Groth16 is based on R1CS arithmetization and PLONK is based on PLONKish arithmetization, which considers a more general class of polynomial equations over finite fields. Variants of PLONK (e.g., HyperPLONK [11], TurboPLONK [12] and UltraPLONK [13]) share the same PLONK style arithmetization, with compatible extensions, which are collectively referred to as PLONKish arithmetization.

zk-STARK [14] is one of the other popular ZK proof systems, whose typical realizations such as Cairo [15] base themselves on AIR⁸ arithmetization. Since AIR is usually considered as a restricted form of PLONKish arithmetization, TABBY can also find itself applicable.

ZK languages and compilers Languages specifically designed for writing ZK programs have been a popular topic for both research and industrial work. They all compile ZK programs into their corresponding ZK components, but with a different focus. Languages like Circom [22], ZoKrates [26], and Noir [24] focus on creating a definition of a higher-level domain-specific interface that abstracts out the tedious and error-prone ZK constraint manipulation, leaving the developers with a set of general-purpose operators for describing

⁸AIR is short for *Algebraic Intermediate Representation*.

their computation. Other languages like Leo [25], Cairo [15] and zkEVM [86, 17] elaborate more on the close binding of ZK language constructs with blockchain platforms (such as Ethereum and Starknet), where more complicated operations are packed into APIs. While many of those languages are built upon Halo2 [23], rather than stacking on top of it, the most recent ZK languages instead seek for a “horizontal” extension by creating ZK virtual machines for general-purpose programming languages. For example, both RISC Zero [28], SP1 [29] and Jolt [30] “decorate” Rust constructs with their corresponding ZK constraints. This shortens the compilation workflow and makes the ZK compiler behind the virtual machine and its optimization a critical and challenging topic.

ZK intermediate representations and compiler optimization Arithmetization is one kind of intermediate representations (IRs) that many ZK compilers directly build upon. For example, Circom [22] compiles its programs into R1CS, and performs constraint-level optimization [16]. For PLONKish arithmetization, [35] uses a predefined set of rewrite rules to manually optimize TurboPLONK circuits, while the static analysis framework proposed in [98] to detect common circuit security vulnerabilities may be adapted to perform simple optimizations. Leo [25] programs get compiled into an IR called Aleo before converting into R1CS. As Aleo preserves high-level semantics, typical compiler optimization techniques such as constant folding and propagation, loop unrolling and function inlining, can all be applied on the application level. There are also architectures that support compiler optimization, such as plookup [87] and CirC [27].

There are also dedicated IRs that enables compiler optimization towards different goals. For example, Noir [24], Chiquito [88] and Juvix [89] are designed for better usability as well as compatibility with specific blockchains. CODA [53] is designed for composing safe ZK circuits. CirC [27] focuses more on compatibility between different ZK languages. As the IRs mentioned focus more on usability and language compatibility, TABBY demonstrates

a focus on performance optimization for ZK circuits.

Program Synthesis While prior work on optimizing ZK circuits has primarily focused on domain-specific compiler techniques, our approach is also deeply informed by advances in program synthesis, especially sketch-based [48, 49] and component-based synthesis [99, 50, 100, 51]. Sketch [48] pioneered the use of partial programs with holes and constraint-based completion via bounded verification, a strategy that underpins TABBY’s synthesis of low-level gadgets from partially specified templates. TABBY leverages similar bounded symbolic evaluation techniques, but applies them in a domain-specific intermediate representation designed for ZK circuits.

TABBY is also inspired by component-based synthesis approaches [99, 50, 51, 101], which construct complex programs by composing reusable subcomponents, often guided by lightweight pruning techniques based on abstract semantics. Systems such as Neo [50] and Morpheus [51] combine symbolic reasoning with search over a component library, leveraging abstract specifications to navigate the synthesis space. TABBY adopts a similar philosophy, decomposing programs into small, equivalence-preserving units and assembling optimized circuits from a growing library of synthesized gadgets. These connections place TABBY at the intersection of ZK-specific circuit optimization and general-purpose program synthesis research.

Recent work has applied program synthesis to cryptography compilers, particularly for homomorphic encryption (HE) circuits [102, 103]. While both ZK circuits and HE circuits share the “circuits” abstraction, they differ substantially in their computation and cost models. For instance, the FHE circuits studied in [102] are simple Boolean circuits with cost models based on circuit multiplicative depth. In contrast, ZK circuits support richer primitives, including large finite fields, equational constraints, and non-determinism, which introduce significantly greater complexity in both reasoning and

cost modeling [42]. Despite these differences, prior work in this area largely relies on established program synthesis techniques such as rewrite-based optimization [102] and sketch-based synthesis [103], the latter of which is closely related to the approach adopted by TABBY.

4.8 Conclusion

We introduced TABBY, a synthesis-aided compiler that brings together high-level programming abstractions and low-level performance optimization for zero-knowledge proof (ZKP) circuits. TABBY is built around a domain-specific intermediate representation that exposes opportunities for arithmetic-level reasoning, and it applies sketch-based program synthesis to automatically discover and verify optimized circuit implementations.

Our approach enables developers to write ZK applications in high-level languages while still benefiting from performance typically reserved for hand-optimized circuits. By decomposing programs into reusable components and verifying equivalence through SMT-based symbolic evaluation, TABBY ensures correctness without sacrificing efficiency.

We evaluated TABBY on a suite of real-world ZKP programs, demonstrating that it consistently reduces constraint counts and proof generation times across different proving backends, including both PLONKish and STARK-based systems. These results illustrate the potential of applying formal techniques—particularly symbolic reasoning and component-based synthesis—within compiler infrastructures for cryptographic applications.

Chapter 5

Conclusion

This dissertation has tackled two persistent obstacles to deploying zero-knowledge proofs at scale: the difficulty of writing *correct* circuits and of compiling them to *efficient* low-level constraints. We argued that they are both tractable through programming-language and compiler techniques. We defended this claim by developing three systems, each targeting a distinct piece of the problem and each evaluated on circuits drawn from real-world ZK projects.

In Chapter 2, we introduced CODA, a statically-typed functional language in which the functional correctness of an arithmetic circuit can be specified through refinement types and statically discharged via a combination of automated proof tactics and interactive verification in Rocq. CODA narrows the gap between a developer’s intended computation and the low-level R1CS constraints that actually realize it: well-typed circuits are guaranteed to satisfy its specification. Re-implementing 77 circuits from 9 Circom projects in CODA, we formally verified 66 of them and uncovered 6 previously unknown vulnerabilities in widely-used codebases.

In Chapter 3 and Chapter 4, we turned to circuit efficiency. TESSEL introduced a tabular intermediate representation that exposes the otherwise-implicit layout of a

PLONKish circuit and reduces the problem of producing an efficient circuit to a precisely stated layout problem on this IR. A partial-evaluation pipeline lowers high-level DSL programs onto the IR, and an eager heuristic produces concrete layouts, yielding measurable speedups over mainstream compilers on a subset of our benchmark suite. TABBY addresses the cases where the heuristic falls short. Reformulating the same optimization problem as program synthesis, TABBY explores the space of equivalent circuits guided by abstract semantics and data-dependency decomposition, producing more consistent reductions in proving and verification costs across a broader range of inputs.

Taken together, these three systems demonstrate that the correctness and efficiency of ZK circuits are not solely the province of expert hand-tuning. Refinement types give developers a lightweight, tightly coupled way to specify and prove what their circuits compute, while compilers that automatically perform domain-specific circuit optimization can recover much of the performance that has historically required manual fine-tuning. Many open problems remain: scaling formal verification to larger circuits and richer specifications, sharing infrastructure across arithmetizations, and integrating correctness and efficiency in a single source language among them. But we hope that the techniques developed here are a useful step toward a future in which writing a trustworthy, efficient zero-knowledge application is as routine as writing any other piece of software.

Appendix A

Supplementary Material for Chapter 2

A.1 Operational Semantics

The full operational semantics for CODA is shown in Figure A.1.

$ \begin{array}{l} v ::= \\ \quad c \\ \quad (v_1, \dots, v_n) \\ \quad v_1 :: v_2 \\ \quad \text{Closure}(\lambda x : \tau. e, \sigma) \\ \sigma ::= \\ \quad \emptyset \\ \quad \sigma[x \mapsto v] \end{array} $	Values: - constants - product value - array value - closure Valuation: - empty valuation - augmentation
$\frac{}{\sigma \vdash v \Downarrow v} \text{E-VALUE}$	$\frac{\sigma(x) = v}{\sigma \vdash x \Downarrow v} \text{E-VAR}$
$\frac{v \in \mathbb{F}}{\sigma \vdash \star \Downarrow v} \text{E-NONDET}$	$\frac{\sigma \vdash e_1 \Downarrow f \quad \sigma \vdash e_2 \Downarrow f \quad f \in \mathbb{F}}{\sigma \vdash \text{assert } e_1 = e_2 \Downarrow \text{unit}} \text{E-ASSERT}$
$\frac{\sigma \vdash e_1 \Downarrow f_1 \quad \sigma \vdash e_2 \Downarrow f_2 \quad f_1 \otimes f_2 = f}{\sigma \vdash e_1 \otimes e_2 \Downarrow f} \text{E-FBINOP}$	
$\frac{\sigma \vdash e_1 \Downarrow z_1 \quad \sigma \vdash e_2 \Downarrow z_2 \quad z_1 \oplus z_2 = z}{\sigma \vdash e_1 \oplus e_2 \Downarrow z} \text{E-INTBINOP}$	
$\frac{\sigma \vdash e_1 \Downarrow b_1 \quad \sigma \vdash e_2 \Downarrow b_2 \quad b_1 \odot b_2 = b}{\sigma \vdash b_1 \odot b_2 \Downarrow b} \text{E-BOOLBINOP}$	
$\frac{\sigma \vdash e_1 \Downarrow v_1 \quad \sigma \vdash e_2 \Downarrow v_2 \quad v_1, v_2 \in \mathbb{F} \text{ or } v_1, v_2 \in \mathbb{Z} \text{ or } v_1, v_2 \in \mathbb{B} \quad v_1 = v_2}{\sigma \vdash e_1 = e_2 \Downarrow \text{true}} \text{E-EQTRUE}$	
$\frac{\sigma \vdash e_1 \Downarrow v_1 \quad \sigma \vdash e_2 \Downarrow v_2 \quad (v_1, v_2) \notin \odot}{\sigma \vdash e_1 \odot e_2 \Downarrow \text{false}} \text{E-EQFALSE}$	
$\frac{\sigma \vdash e_1 \Downarrow z_1 \quad \sigma \vdash e_2 \Downarrow z_2 \quad v_1, v_2 \in \mathbb{Z} \quad \odot \in \{<, \leq\} \quad (z_1, z_2) \in \odot}{\sigma \vdash e_1 \odot e_2 \Downarrow \text{true}} \text{E-INEQTRUE}$	
$\frac{\sigma \vdash e_1 \Downarrow v_1 \quad \sigma \vdash e_2 \Downarrow v_2 \quad v_1, v_2 \in \mathbb{Z} \quad \odot \in \{<, \leq\} \quad (z_1, z_2) \notin \odot}{\sigma \vdash e_1 \odot e_2 \Downarrow \text{false}} \text{E-INEQFALSE}$	

Figure A.1: Semantics of CODA program

$$\begin{array}{c}
\frac{}{\sigma \vdash \lambda x : _ . e \Downarrow \text{Closure}(\lambda x : _ . e, \sigma)} \text{E-LAM} \\
\\
\frac{\sigma \vdash e_1 \Downarrow \text{Closure}(\lambda x : _ . e, \sigma') \quad \sigma \vdash e_2 \Downarrow v \quad \sigma'[x \mapsto v] \vdash e \Downarrow v'}{\sigma \vdash e_1 e_2 \Downarrow v'} \text{E-APP} \\
\\
\frac{\sigma \vdash e_1 \Downarrow v_1 \quad \sigma \vdash e_2 \Downarrow v_2}{\sigma \vdash e_1 :: e_2 \Downarrow v_1 :: v_2} \text{E-CONS} \quad \frac{\sigma \vdash e_1 \Downarrow v_h :: v_t \quad \sigma \vdash e_2 \Downarrow 0}{\sigma \vdash e_1 [e_2] \Downarrow v_h} \text{E-INDCONSHHEAD} \\
\\
\frac{\sigma \vdash e_1 \Downarrow v_h :: v_t \quad \sigma \vdash e_2 \Downarrow z \quad 0 < z \quad \sigma \vdash v_t [z - 1] \Downarrow v}{\sigma \vdash e_1 [e_2] \Downarrow v} \text{E-INDCONSTAIL} \\
\\
\frac{\sigma \vdash e_i \Downarrow v_i \text{ for all } i \in \{1, \dots, n\}}{\sigma \vdash (e_1, \dots, e_i) \Downarrow (v_1, \dots, v_i)} \text{E-PRODCONS} \\
\\
\frac{\sigma \vdash e_1 \Downarrow (v_1, \dots, v_n) \quad \sigma[x_1 \mapsto v_1] \dots [x_n \mapsto v_n] \vdash e_2 \Downarrow v}{\sigma \vdash \text{match } e_1 \text{ with } (x_1, \dots, x_n) \rightarrow e_2 \Downarrow v} \text{E-PRODDESTR} \\
\\
\frac{\sigma \vdash e_l \Downarrow v_l \text{ for all } l \in \{s, e, f, a\} \quad v_s \in \mathbb{Z} \quad v_e \in \mathbb{Z} \quad v_s \geq v_e}{\sigma \vdash \text{iter}_{\lambda i. \tau} e_s e_e e_f e_a \Downarrow v_a} \text{E-ITER0} \\
\\
\frac{v_s \in \mathbb{Z} \quad v_e \in \mathbb{Z} \quad v_s < v_e \quad \sigma \vdash \text{iter}_{\lambda i. \tau} (v_s +_{\mathbb{Z}} 1) v_e v_f (v_f v_a) \Downarrow v}{\sigma \vdash \text{iter}_{\lambda i. \tau} e_s e_e e_f e_a \Downarrow v} \text{E-ITERS} \\
\\
\frac{\Delta(C) = \mathbf{circuit} \ R \ (x_1 : T_1) \dots (x_n : T_n) \rightarrow T \ \{e\} \quad \sigma \vdash e_i \Downarrow v_i \text{ for all } i \in \{1, \dots, n\} \quad \sigma[x_1 \mapsto v_1] \dots [x_n \mapsto v_n] \vdash e \Downarrow v}{\sigma \vdash \#C \ e_1 \dots e_n \Downarrow v} \text{E-CREF}
\end{array}$$

Figure A.1: Semantics of CODA program (continued)

A.2 Metatheory

In this section, we include some of the metatheory about CODA's type system as well as a brief sketch of the proof of Theorem 2.1.

For the purposes of the proofs, we consider a core subset of the calculus that contains only field elements, integers, booleans, pairs (products of arity 2), functions, and iter expressions. Furthermore, we take "unit" to be a first-class expression `unit` and type `Unit`.

A.2.1 Preliminary Definitions

Definition A.1. The valuation typing relation, written as $\Delta; \Gamma \models \sigma$, is defined as:

$$\begin{aligned}
 \Delta; \Gamma \models \sigma ::= & \\
 & \text{dom}(\Gamma) = \text{dom}(\sigma) \\
 \wedge \quad & (\forall x, v, \tau. \\
 & \quad \Gamma(x) = \{\nu : T \mid \phi\} \wedge \sigma(x) = v \implies \\
 & \quad \Delta; \Gamma \vdash v : \{\nu : T \mid \nu = x\}) \\
 \wedge \quad & (\forall x, v, x', \tau_1, \tau_2. \\
 & \quad \Gamma(x) = x' : \tau_1 \rightarrow \tau_2 \wedge \sigma(x) = v \implies \\
 & \quad \Delta; \Gamma \vdash v : x' : \tau_1 \rightarrow \tau_2)
 \end{aligned}$$

The meaning of valuation typing is "a variable x is in Γ if and only if it is also in σ ; and the value v of every variable x in σ must have the same type as x under Γ ." Note that the two "cases" here correspond to the types in the TE-VAR and TE-VAR-FUN rules, respectively.

Definition A.2. The base type subtyping judgment, written as $\models T_1 <: T_2$, is defined using the inference rules in fig. A.2.

A.2.2 Subtyping Lemmas

We now state several useful properties about our subtyping rules.

$$\begin{array}{c}
 \frac{}{\models T <: T} \text{BSUB-REFL} \\
 \\
 \frac{\models T_{11} <: T_{21} \quad \models T_{12} <: T_{22}}{\models (T_{11}, T_{12}) <: (T_{21}, T_{22})} \text{BSUB-PAIR}
 \end{array}$$

Figure A.2: Base Type Subtyping Rules

Lemma A.3 (Every Refinement Type is a Subtype of True). *For all Γ , T , and ϕ ,*

$$\Gamma \models \{\nu : T \mid \phi\} <: \{\nu : T \mid \text{true}\} \quad (\text{A.1})$$

Proof. By induction on the derivation of eq. (A.1). $\square$

Lemma A.4 (Inversion for Function Subtyping, Unknown Supertype). *If $\Gamma \models x : \tau_1 \rightarrow \tau_2 <: \tau$, then there exist τ'_1, τ'_2 such that $\tau = x : \tau'_1 \rightarrow \tau'_2$ and $\Gamma \models \tau'_1 <: \tau_1$ and $\Gamma \models \tau_2 <: \tau'_2$.*

Proof. By induction on the derivation of $\Gamma \models x : \tau_1 \rightarrow \tau_2 <: \tau$. $\square$

Lemma A.5 (Inversion for Function Subtyping, Both Known). *If $\Gamma \models x : \tau_1 \rightarrow \tau_2 <: y : \tau'_1 \rightarrow \tau'_2$, then $x = y$ and $\Gamma \models \tau'_1 <: \tau_1$ and $\Gamma \models \tau_2 <: \tau'_2$.*

Proof. By induction on the derivation of $\Gamma \models x : \tau_1 \rightarrow \tau_2 <: y : \tau'_1 \rightarrow \tau'_2$. $\square$

Lemma A.6 (Inversion for Refinement Subtyping). *If $\Gamma \models \{\nu : T_1 \mid p\} <: \tau$, then there exists a T'_2, q such that all of the following hold:*

- $\tau = \{\nu : T_2 \mid q\}$
- $\Gamma \models T_1 <: T_2$
- $\models_{\text{Coq}} \forall \vec{x} \in \text{dom}(\Gamma). \forall \nu. \text{Encode}(\Gamma) \implies \llbracket p \rrbracket \implies \llbracket q \rrbracket$

Proof. By induction on the derivation of $\Gamma \models \{\nu : T_1 \mid p\} <: \tau$. $\square$

Lemma A.7 (Subtyping Weakening). *If $\Delta; \Gamma \models \tau_1 <: \tau_2$ and $x \notin \text{dom}(\Gamma)$, then $\Delta; \Gamma, x : \tau_x \models \tau_1 <: \tau_2$.*

Proof. By induction on the derivation of $\Delta; \Gamma \models \tau_1 <: \tau_2$. $\square$

A.2.3 Expression Typing Lemmas

Lemma A.8 (Expression Typing Weakening). *If $\Delta; \Gamma \vdash e : \tau$ and $x \notin \text{dom}(\Gamma)$, then $\Delta; \Gamma, x : \tau_x \vdash e : \tau$*

Proof. By induction on the derivation of $\Delta; \Gamma \vdash e : \tau$. $\square$

Lemma A.9 (Inversion for Types). *Suppose that $\Gamma \vdash e : \tau$. Then there exists a type τ' such that:*

$$\Delta; \Gamma \vdash e : \tau' \quad (\text{A.2})$$

$$\Gamma \models \tau' <: \tau \quad (\text{A.3})$$

and:

- (a) If $e = f$ for some $f \in \mathbb{F}_p$, then $\tau' = \{\nu : F \mid \nu = f\}$.
- (b) If $e = b$ for some $b \in \{\text{true}, \text{false}\}$, then $\tau' = \{\nu : \text{Bool} \mid \nu = b\}$.
- (c) If $e = x$, then exactly one of the following is true:
- There exist T, ϕ such that $\tau' = \tau' = \{\nu : T \mid \nu = x\}$ and $\Gamma(x) = \{\nu : T \mid \phi\}$.
 - There exist $x', \tau_1, \tau_2, \tau'_1, \tau'_2$ such that $\tau = x' : \tau_1 \rightarrow \tau_2$ and $\Gamma(x) = x' : \tau'_1 \rightarrow \tau'_2$ and $\Delta; \Gamma \vdash x : x' : \tau'_1 \rightarrow \tau'_2$.
- (d) If $e = \star$, then $\tau' = \{\nu : F \mid \text{true}\}$.
- (e) If $e = \text{Closure}(\lambda x : \tau'_1. e', \sigma)$ for some e', τ'_1 , then there exist $\tau_1, \tau_2, \tau'_2, \Gamma'$ such that $\tau = x : \tau_1 \rightarrow \tau_2$ and $\tau' = x : \tau'_1 \rightarrow \tau'_2$ and $\Delta; \Gamma', x : \tau'_1 \vdash e : \tau'_2$ and $\text{FV}(\tau'_2) \subseteq \{\nu, x\}$.
- (f) If $e = \lambda x : \tau_1. e'$, then there exists e, τ_2 such that $\Delta; \Gamma \vdash (\lambda x : \tau_1. e) : x : \tau_1 \rightarrow \tau_2$.
- (g) If $e = e_1 e_2$, then there exist x, τ'_1, τ'_2 such that $\tau' = \tau'_2[e_2 \mapsto x]$ and $\Delta; \Gamma \vdash e_1 : x : \tau'_1 \rightarrow \tau'_2$ and $\Delta; \Gamma \vdash e_1 e_2 : \tau'$.
- (h) If $e = (e_1, e_2)$, then there exists $T_1, T_2, \phi, T'_1, T'_2, \phi_1, \phi_2$ such that:

$$\tau = \{\nu : (T_1, T_2) \mid \phi\} \quad (\text{A.4})$$

$$\tau' = \{\nu : (T'_1, T'_2) \mid \phi_1[\nu \mapsto \nu.1] \wedge \phi_2[\nu \mapsto \nu.2]\} \quad (\text{A.5})$$

$$\Gamma \models \tau' <: \tau \quad (\text{A.6})$$

$$\Delta; \Gamma \vdash (e_1, e_2) : \tau' \quad (\text{A.7})$$

$$\Delta; \Gamma \vdash e_1 : \{\nu : T'_1 \mid \phi_1\} \quad (\text{A.8})$$

$$\Delta; \Gamma \vdash e_2 : \{\nu : T'_2 \mid \phi_2\} \quad (\text{A.9})$$

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$. As there are many similar cases, we show only some of the cases here.

- Case TE-CONSTF: Then we have

$$e = f \quad (\text{A.10})$$

$$\tau = \{\nu : F \mid \nu = f\} \quad (\text{A.11})$$

Choosing $\tau' = \tau$ discharges eq. (A.2) and clause (a), and then applying TSUB-REFL discharges eq. (A.3).

- Cases TE-NONDET, TE-CONSTBOOL, TE-VAR, TE-ABS: Similar.

- Case TE-SUB: Then

$$\Gamma \models \tau'' <: \tau \quad (\text{A.12})$$

$$\Delta; \Gamma \vdash e : \tau'' \quad (\text{A.13})$$

The inductive hypothesis is:

$$\Gamma \vdash e : \tau'' \implies \quad (\text{A.14})$$

$$\exists \tau'.$$

$$\Delta; \Gamma \vdash e : \tau' \wedge \quad (\text{A.15})$$

$$\Gamma \models \tau' <: \tau'' \wedge \quad (\text{A.16})$$

$$\text{all clauses (a) - (h)} \quad (\text{A.17})$$

Clearly eq. (A.14) is satisfied. Thus eq. (A.2) and eq. (A.3) follow immediately (the latter through transitivity of eq. (A.16) and eq. (A.12)).

We then proceed by case analysis on e to show the remaining clauses.

- Case $e = f$: Only clause (a) applies, and eq. (A.17) implies that as required,

$$\tau' = \{\nu : F \mid \nu = f\} \quad (\text{A.18})$$

- Case $e = \star$: Only clause (d) applies, and eq. (A.17) implies that as required,

$$\tau' = \{\nu : F \mid \text{true}\} \quad (\text{A.19})$$

- The remaining cases are similar.

□

Lemma A.10. *If $\Delta; \Gamma \vdash e : \tau$, then for all x, τ' , we have $\Delta; \Gamma, x : \tau' \vdash e : \tau$.*

Proof. By induction on the derivation of $\Delta; \Gamma \vdash e : \tau$. □

A.2.4 Correctness Properties

Theorem 2.1 (Expression Type Preservation). *If $\Delta; \Gamma \vdash e : \tau$ and $\sigma \vdash e \Downarrow v$ and $\Delta; \Gamma \models \sigma$, then $\Delta; \Gamma \vdash v : \tau$.*

Proof. By induction on the derivation of $\sigma \vdash e \Downarrow v$, where we want to prove the property:

$$\forall \Gamma, \tau. \quad \Gamma \vdash e : \tau \implies \Gamma \vdash v : \tau$$

As there are many cases, we show only some of the cases here.

- Case E-VALUE: Follows immediately from the premises.

- Case E-NONDET: Then

$$e = \star \tag{A.20}$$

$$v = f \in \mathbb{F}_p \tag{A.21}$$

From Lemma A.9,

$$\Gamma \models \{\nu : F \mid \text{true}\} <: \tau \tag{A.22}$$

$$\Delta; \Gamma \vdash e : \{\nu : F \mid \text{true}\} \tag{A.23}$$

Applying TE-CONSTF and Lemma A.3 to v then yields the desired result.

- Case E-VAR: Then

$$e = x \tag{A.24}$$

$$v = \sigma(x) \tag{A.25}$$

By Lemma A.9, we know that τ is either a refinement type or a function.

If τ is a refinement type, i.e. $\Gamma(x) = \{\nu : T \mid \phi\}$ for some T, ϕ , then we have that $\Gamma \models \{\nu : T \mid \nu = x\} <: \tau$. From Definition A.1, it follows that $\Gamma \vdash v : \{\nu : T \mid \nu = x\}$. The conclusion then follows by applying the TE-SUB rule.

Otherwise, the τ is a function type, i.e. $\Gamma(x) = x' : \tau_1 \rightarrow \tau_2$ for some x', τ_1, τ_2 . The conclusion follows similarly in this case.

- Case E-ASSERT: Then

$$e = (\text{assert } x_1 = x_2) \tag{A.26}$$

$$\sigma \vdash x_1 \Downarrow f \tag{A.27}$$

$$\sigma \vdash x_2 \Downarrow f \tag{A.28}$$

$$v = \text{unit} \tag{A.29}$$

In order to show that $\Delta; \Gamma \vdash \text{unit} : \{\nu : \text{Unit} \mid x_1 = x_2\}$, we can use the TE-SUB, TE-UNIT, and TSUB-REFINE rules. This requires us to show that $x_1 = x_2$, when embedded as a Coq proposition, is entailed by the embedding of Γ :

$$\begin{aligned} & \models_{\text{Coq}} \forall \vec{x} \in \text{dom}(\Gamma). \forall \nu. \\ & \text{Encode}(\Gamma) \implies \\ & \llbracket \nu = \text{unit} \rrbracket \implies \llbracket x_1 = x_2 \rrbracket \end{aligned} \tag{A.30}$$

We now argue that the above proposition holds. First, by Lemma A.9, we have that $\Delta; \Gamma \vdash x_1 : \{\nu : F \mid \nu = x_1\}$ (similarly for x_2). Using the inductive hypotheses for x_1

and x_2 , it can then be shown, resp., that

$$\Delta; \Gamma \vdash f : \{\nu : F \mid \nu = x_1\} \quad (\text{A.31})$$

$$\Delta; \Gamma \vdash f : \{\nu : F \mid \nu = x_2\} \quad (\text{A.32})$$

Applying Lemma A.9 to the above then yields

$$\Gamma \models \{\nu : F \mid \nu = f\} <: \{\nu : F \mid \nu = x_1\} \quad (\text{A.33})$$

$$\Gamma \models \{\nu : F \mid \nu = f\} <: \{\nu : F \mid \nu = x_2\} \quad (\text{A.34})$$

By Lemma A.6, we must have

$$\begin{aligned} & \models_{\text{Coq}} \forall \vec{x} \in \text{dom}(\Gamma). \forall \nu. \\ & \quad \text{Encode}(\Gamma) \implies \\ & \quad \llbracket \nu = f \rrbracket \implies \llbracket \nu = x_1 \rrbracket \end{aligned} \quad (\text{A.35})$$

$$\begin{aligned} & \models_{\text{Coq}} \forall \vec{x} \in \text{dom}(\Gamma). \forall \nu. \\ & \quad \text{Encode}(\Gamma) \implies \\ & \quad \llbracket \nu = f \rrbracket \implies \llbracket \nu = x_2 \rrbracket \end{aligned} \quad (\text{A.36})$$

If we instantiate ν in both propositions by setting $\nu = f$, then together they will imply the single proposition:

$$\begin{aligned} & \models_{\text{Coq}} \forall \vec{x} \in \text{dom}(\Gamma). \\ & \quad \text{Encode}(\Gamma) \implies \\ & \quad \llbracket f = f \rrbracket \implies \llbracket f = x_1 \wedge f = x_2 \rrbracket \end{aligned} \quad (\text{A.37})$$

Finally, the above proposition implies that:

$$\begin{aligned} & \models_{\text{Coq}} \forall \vec{x} \in \text{dom}(\Gamma). \\ & \quad \text{Encode}(\Gamma) \implies \\ & \quad \llbracket \text{true} \rrbracket \implies \llbracket x_1 = x_2 \rrbracket \end{aligned} \quad (\text{A.38})$$

This implies eq. (A.30), which completes the proof of this case.

- Case E-PAIR: Then we have

$$e = (e_1, e_2) \quad (\text{A.39})$$

$$v = (v_1, v_2) \quad (\text{A.40})$$

First, from Lemma A.9, we have

$$\tau = \{\nu : (T_1, T_2) \mid \phi\} \quad (\text{A.41})$$

$$\tau' = \{\nu : (T'_1, T'_2) \mid \phi_1[\nu \mapsto \nu.1] \wedge \phi_2[\nu \mapsto \nu.2]\} \quad (\text{A.42})$$

$$\Gamma \models \tau' <: \tau \quad (\text{A.43})$$

$$\Delta; \Gamma \vdash (e_1, e_2) : \tau' \quad (\text{A.44})$$

$$\Delta; \Gamma \vdash e_1 : \{\nu : T'_1 \mid \phi_1\} \quad (\text{A.45})$$

$$\Delta; \Gamma \vdash e_2 : \{\nu : T'_2 \mid \phi_2\} \quad (\text{A.46})$$

We then apply the inductive hypotheses with e_1 and e_2 to obtain:

$$\Delta; \Gamma \vdash v_1 : \{\nu : T'_1 \mid \phi_1\} \quad (\text{A.47})$$

$$\Delta; \Gamma \vdash v_2 : \{\nu : T'_2 \mid \phi_2\} \quad (\text{A.48})$$

Finally, we apply TE-SUB and TE-PAIR to derive the conclusion.

- Cases E-FBINOP, E-BBINOP, E-INTBINOP: Similar.
- Case E-APP: Then

$$e = e_1 \ e_2 \quad (\text{A.49})$$

$$\sigma \vdash e_1 \Downarrow \text{Closure}(\lambda x : \tau'_1. e', \sigma) \quad (\text{A.50})$$

$$\sigma \vdash e_1 \Downarrow v_2 \quad (\text{A.51})$$

$$\sigma[x \mapsto v_2] \vdash e' \Downarrow v \quad (\text{A.52})$$

By Lemma A.9 on the type of e , there must exist $x', \tau_1, \tau_2, \tau'_2$ such that

$$\Delta; \Gamma \vdash e_1 : x' : \tau_1 \rightarrow \tau_2 \quad (\text{A.53})$$

$$\Gamma \models x' : \tau'_1 \rightarrow \tau'_2 <: \tau \quad (\text{A.54})$$

$$\Gamma \models \tau_2[x' \mapsto e_2] <: \tau \quad (\text{A.55})$$

Applying the inductive hypothesis to e_1 then yields

$$\Delta; \Gamma \vdash \text{Closure}(\lambda x : \tau'. e', \sigma) : x' : \tau'_1 \rightarrow \tau'_2 \quad (\text{A.56})$$

Now, by applying Lemma A.9 to the closure, it follows that there exist Γ' and τ''_2 such that

$$\Delta; \Gamma \vdash \text{Closure}(\lambda x : \tau. e', \sigma) : x' : \tau'_1 \rightarrow \tau''_2 \quad (\text{A.57})$$

$$\Gamma \models x' : \tau'_1 \rightarrow \tau''_2 <: x' : \tau'_1 \rightarrow \tau'_2 \quad (\text{A.58})$$

$$\Delta; \Gamma', x : \tau'_1 \vdash e' : \tau'' \quad (\text{A.59})$$

The inductive hypothesis can then be applied to e' to obtain:

$$\Delta; \Gamma', x : \tau'_1 \vdash v : \tau'' \quad (\text{A.60})$$

Note that τ'' may only contain x' and ν as free variables; consequently, $\tau''[x' \mapsto e_2]$ contains only free variables in $\text{dom}(\Gamma)$. Furthermore, because v is already well-typed under Γ' and has no free variables besides ν , it can be typed under any other typing context. It follows that $\Delta; \Gamma \vdash v : \tau''[x' \mapsto e_2]$. The desired result follows immediately by applying TE-SUB and eq. (A.55).

- Case E-LAM: Follows from Lemma A.9, TE-SUB, and TE-CLO.
- Cases E-CREF, E-ITER0, E-ITERS: Similar to E-APP.

□

Bibliography

- [1] S. Goldwasser, S. Micali, and C. Rackoff, *The knowledge complexity of interactive proof-systems (extended abstract)*, in *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA* (R. Sedgwick, ed.), pp. 291–304, ACM, 1985.
- [2] E. Ben Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, *Zerocash: Decentralized anonymous payments from bitcoin*, in *2014 IEEE Symposium on Security and Privacy*, pp. 459–474, 2014.
- [3] M. labs, “The future-proof zkevm.” <https://zksync.io/>, 2022.
- [4] Scroll, “The native zkevm scaling solution for ethereum.” <https://scroll.io/>, 2022.
- [5] Polygon, “Bring ethereum to everyone.” <https://polygon.technology/polygon-zkevm>, 2022.
- [6] F. Baldimtsi, K. K. Chalkias, Y. Ji, J. Lindstrøm, D. Maram, B. Riva, A. Roy, M. Sedaghat, and J. Wang, *zklogin: Privacy-preserving blockchain authentication with existing credentials*, in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS ’24*, (New York, NY, USA), p. 3182–3196, Association for Computing Machinery, 2024.
- [7] Google, “longfellow-zk.” <https://github.com/google/longfellow-zk>, 2025.
- [8] B.-J. Chen, S. Waiwitlikhit, I. Stoica, and D. Kang, *Zkml: An optimizing system for ml inference in zero-knowledge proofs*, in *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys ’24*, (New York, NY, USA), p. 560–574, Association for Computing Machinery, 2024.
- [9] J. Groth, *On the size of pairing-based non-interactive arguments*, in *Advances in Cryptology – EUROCRYPT 2016* (M. Fischlin and J.-S. Coron, eds.), (Berlin, Heidelberg), pp. 305–326, Springer Berlin Heidelberg, 2016.
- [10] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, “PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge.” *Cryptology ePrint Archive*, Paper 2019/953, 2019.

- [11] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, *Hyperplonk: Plonk with linear-time prover and high-degree custom gates*, in *Advances in Cryptology – EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23–27, 2023, Proceedings, Part II*, (Berlin, Heidelberg), p. 499–530, Springer-Verlag, 2023.
- [12] A. Gabizon and Z. J. Williamson, *The turbo-plonk program syntax for specifying snark programs*, in *Proceedings of the 3rd ZKProof Workshop*, 2020.
- [13] Z. Williamson, “Aztec’s ZK-ZK-rollup, looking behind the cryptocurtain.” <https://medium.com/aztec-protocol/aztecs-zk-zk-rollup-looking-behind-the-cryptocurtain-2b8af1fca619>, Mar., 2021. Accessed: 2024-9-10.
- [14] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, *Scalable, transparent, and post-quantum secure computational integrity*, *IACR Cryptol. ePrint Arch.* (2018) 46.
- [15] L. Goldberg, S. Papini, and M. Riabzev, *Cairo - a turing-complete stark-friendly cpu architecture*, *IACR Cryptol. ePrint Arch.* **2021** (2021) 1063.
- [16] E. Albert, M. Bellés-Muñoz, M. Isabel, C. Rodríguez-Núñez, and A. Rubio, *Distilling constraints in zero-knowledge protocols*, in *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I*, (Berlin, Heidelberg), p. 430–443, Springer-Verlag, 2022.
- [17] Scroll, “Scroll zkEVM.” <https://github.com/scroll-tech/zkevm-circuits>. Accessed: 2024-9-10.
- [18] trailofbits, “The frozen heart vulnerability in bulletproofs.” <https://tinyurl.com/2z7nw5ku>, 2022.
- [19] Aztec, “Disclosure of recent vulnerabilities.” <https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities>, 2022.
- [20] electriccoin, “Zcash counterfeiting vulnerability successfully remediated.” <https://tinyurl.com/3sv793fj>, 2019.
- [21] TornadoCash, “Tornado.cash got hacked. by us.” <https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8>, 12, 2019.
- [22] M. Bellés-Muñoz, J. Baylina, V. Daza, and J. L. Muñoz-Tapia, *New privacy practices for blockchain software*, *IEEE Software* **39** (2022), no. 3 43–49.
- [23] Zcash, “halo2.” <https://github.com/zcash/halo2>. Accessed: 2024-9-10.
- [24] Aztec, “Noir.” <https://github.com/noir-lang/noir>. Accessed: 2024-9-10.

- [25] C. Chin, “Leo: A programming language for formally verified, zero-knowledge applications.” <https://docs.zkproof.org/pages/standards/accepted-workshop4/proposal-leo.pdf>, 2021.
- [26] J. Eberhardt and S. Tai, *Zokrates - scalable privacy-preserving off-chain computations*, in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 1084–1091, 2018.
- [27] A. Ozdemir, F. Brown, and R. S. Wahby, “Circ: Compiler infrastructure for proof systems, software verification, and more.” Cryptology ePrint Archive, Paper 2020/1586, 2020. <https://eprint.iacr.org/2020/1586>.
- [28] RISC Zero, “RISC zero.” <https://github.com/risc0/risc0>. Accessed: 2024-9-10.
- [29] Succinct Labs, “SP1.” <https://github.com/succinctlabs/sp1>. Accessed: 2024-9-10.
- [30] A. Arun, S. Setty, and J. Thaler, “Jolt: SNARKs for virtual machines via lookups.” Cryptology ePrint Archive, Paper 2023/1217, 2023.
- [31] Trail of Bits, “Circomspect.” <https://github.com/trailofbits/circomspect>.
- [32] S. Pailoor, Y. Chen, F. Wang, C. Rodríguez, J. Van Geffen, J. Morton, M. Chu, B. Gu, Y. Feng, and I. Dillig, *Automated detection of under-constrained circuits in zero-knowledge proofs*, *Proc. ACM Program. Lang.* **7** (June, 2023).
- [33] H. Wen, J. Stephens, Y. Chen, K. Ferles, S. Pailoor, K. Charbonnet, I. Dillig, and Y. Feng, *Practical security analysis of zero-knowledge proof circuits*, in *Proceedings of the 33rd USENIX Conference on Security Symposium, SEC ’24, (USA)*, USENIX Association, 2024.
- [34] C. Fournet, C. Keller, and V. Laporte, *A certified compiler for verifiable computing*, in *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pp. 268–280, IEEE Computer Society, 2016.
- [35] M. Ambrona, A.-L. Schmitt, R. R. Toledo, and D. Willems, “New optimization techniques for PlonK’s arithmetization.” Cryptology ePrint Archive, Paper 2022/462, 2022.
- [36] Aztec, “Introducing noir.” <https://noir-lang.org/>, 2022.

- [37] P. M. Rondon, M. Kawaguchi, and R. Jhala, *Liquid types*, in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), p. 159–169, Association for Computing Machinery, 2008.
- [38] P. M. Rondon, M. Kawaguchi, and R. Jhala, *Low-level liquid types*, in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, (New York, NY, USA), p. 131–144, Association for Computing Machinery, 2010.
- [39] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, *Refinement types for haskell*, in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, (New York, NY, USA), p. 269–282, Association for Computing Machinery, 2014.
- [40] P. Vekris, B. Cosman, and R. Jhala, *Refinement types for typescript*, in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), p. 310–325, Association for Computing Machinery, 2016.
- [41] B. Tan, B. Mariano, S. K. Lahiri, I. Dillig, and Y. Feng, *Soltype: refinement types for arithmetic overflow in solidity*, *Proc. ACM Program. Lang.* **6** (2022), no. POPL 1–29.
- [42] A. Ozdemir, G. Kremer, C. Tinelli, and C. W. Barrett, *Satisfiability modulo finite fields*, in *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II* (C. Enea and A. Lal, eds.), vol. 13965 of *Lecture Notes in Computer Science*, pp. 163–186, Springer, 2023.
- [43] T. Hader, *Non-linear smt-reasoning over finite fields*, 2022.
- [44] L. C. Vella and L. Alt, *On satisfiability of polynomial equations over large prime fields*, in *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022* (D. Déharbe and A. E. J. Hyvärinen, eds.), vol. 3185 of *CEUR Workshop Proceedings*, pp. 114–127, CEUR-WS.org, 2022.
- [45] A. Ozdemir, S. Pailoor, A. Bassa, K. Ferles, C. Barrett, and I. Dillig, *Split gröbner bases for satisfiability modulo finite fields*, in *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I*, (Berlin, Heidelberg), p. 3–25, Springer-Verlag, 2024.
- [46] E. Pertseva, A. Ozdemir, S. Pailoor, A. Bassa, S. Porncharoenwase, I. Dillig, and C. Barrett, *Integer reasoning modulo different constants in smt*, in *Computer Aided*

- Verification: 37th International Conference, CAV 2025, Zagreb, Croatia, July 23–25, 2025, Proceedings, Part I*, (Berlin, Heidelberg), p. 339–362, Springer-Verlag, 2025.
- [47] INRIA, “The coq proof assistant.” <https://coq.inria.fr/>, 2022.
 - [48] A. Solar-Lezama, *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.
 - [49] E. Torlak and R. Bodik, *A lightweight symbolic virtual machine for solver-aided host languages*, in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’14, (New York, NY, USA), p. 530–541, Association for Computing Machinery, 2014.
 - [50] Y. Feng, R. Martins, O. Bastani, and I. Dillig, *Program synthesis using conflict-driven learning*, in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, (New York, NY, USA), p. 420–435, Association for Computing Machinery, 2018.
 - [51] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri, *Component-based synthesis of table consolidation and transformation tasks from examples*, in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, (New York, NY, USA), p. 422–436, Association for Computing Machinery, 2017.
 - [52] A. Fiat and A. Shamir, *How to prove yourself: Practical solutions to identification and signature problems*, in *Advances in Cryptology — CRYPTO ’86*, vol. 263 of *Lecture Notes in Computer Science*, pp. 186–194, Springer, 1987.
 - [53] J. Liu, I. Kretz, H. Liu, B. Tan, J. Wang, Y. Sun, L. Pearson, A. Miltner, I. Dillig, and Y. Feng, *Certifying zero-knowledge circuits with refinement types*, in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 1741–1759, 2024.
 - [54] J. Liu, J. Song, Y. Chen, H. Liu, H. Wen, L. Pearson, Y. Chen, and Y. Feng, *Tabby: A synthesis-aided compiler for high-performance zero-knowledge proof circuits*, *Proceedings of the ACM on Programming Languages* **9** (2025), no. OOPSLA2.
 - [55] D. Benarroch, K. Gurkan, R. Kahat, A. Nicolas, and E. Tromer, “zkinterface, a standard tool for zero-knowledge interoperability.” <https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-zkinterface.pdf>, 2019.
 - [56] Veridise, “Circom-pairing: A million-dollar zk bug caught early.” <https://tinyurl.com/3u5973pf>, 2023.
 - [57] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, *Simple high-level code for cryptographic arithmetic - with proofs, without compromises*, in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19–23, 2019*, pp. 1202–1219, IEEE, 2019.

- [58] Iden3, “Snarkjs.” <https://github.com/iden3/snarkjs>, 2018.
- [59] E. Albert, M. Bellés-Muñoz, M. Isabel, C. Rodríguez-Núñez, and A. Rubio, *Distilling constraints in zero-knowledge protocols*, in *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I* (S. Shoham and Y. Vizel, eds.), vol. 13371 of *Lecture Notes in Computer Science*, pp. 430–443, Springer, 2022.
- [60] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, *Succinct non-interactive zero knowledge for a von neumann architecture*, in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014* (K. Fu and J. Jung, eds.), pp. 781–796, USENIX Association, 2014.
- [61] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, *Succinct Non-Interactive zero knowledge for a von neumann architecture*, in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 781–796, USENIX Association, Aug., 2014.
- [62] Matter-Labs, “Zinc.” <https://github.com/matter-labs/zinc>, 2022.
- [63] o1 Labs, “Snarky: Write efficient, beautiful, safe zk-snark code.” <https://o1-labs.github.io/snarky/>, 2022.
- [64] N. Amin, J. Burnham, F. Garillot, R. Gennaro, D. Rogozin, C. Wong, *et. al.*, *Lurk: Lambda, the ultimate recursive knowledge*, *Cryptology ePrint Archive* (2023).
- [65] N. Luo, T. Antonopoulos, W. R. Harris, R. Piskac, E. Tromer, and X. Wang, *Proving UNSAT in zero knowledge*, in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022* (H. Yin, A. Stavrou, C. Cremers, and E. Shi, eds.), pp. 2203–2217, ACM, 2022.
- [66] X. Leroy, *Formal verification of a realistic compiler*, *Commun. ACM* **52** (2009), no. 7 107–115.
- [67] R. Corin and J. den Hartog, *A probabilistic hoare-style logic for game-based cryptographic proofs (extended version)*, 2005. To appear in ICALP 2006 Track C corin@cs.utwente.nl 13264 received 23 Dec 2005, last revised 26 Apr 2006.
- [68] M. Gagné, P. Lafourcade, and Y. Lakhnech, “Automated security proofs for almost-universal hash for mac verification.” *Cryptology ePrint Archive*, Paper 2013/407, 2013. <https://eprint.iacr.org/2013/407>.
- [69] A. Tiwari, A. Gascon, and B. Dutertre, *Program synthesis using dual interpretation*, in *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, vol. 9195 of *LNCS*, pp. 482–497, 2015.

- [70] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub, *Easycrypt: A tutorial*, in *FOSAD*, 2013.
- [71] N. Sidorenco, S. Oechsner, and B. Spitters, *Formal security analysis of mpc-in-the-head zero-knowledge protocols*, in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pp. 1–14, 2021.
- [72] J. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider, *A certifying compiler for zero-knowledge proofs of knowledge based on sigma-protocols.*, vol. 6345, pp. 151–167, 09, 2010.
- [73] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/hol: A proof assistant for higher-order logic*, 2002.
- [74] Rust, *A language empowering everyone to build reliable and efficient software*, 2024.
- [75] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, “Scalable, transparent, and post-quantum secure computational integrity.” Cryptology ePrint Archive, Paper 2018/046, 2018.
- [76] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers principles, techniques & tools*. pearson Education, 2007.
- [77] Y. Zhang, *Plonk cost model*, 2022. Accessed: 2024-9-12.
- [78] D. Griswold, *The java hotspot virtual machine architecture*, *Sun Microsystems Whitepaper* (1998).
- [79] Axiom, *Compute over the history of ethereum in your smart contract*, 2024.
- [80] Dhole, *polyexen*, 2022.
- [81] Noir, *Awesome noir*, 2024.
- [82] “Halo2_backend.” https://github.com/Ethan-000/halo2_backend.
- [83] PSE, *Privacy + scaling explorations*, 2024.
- [84] S. Bowe, J. Grigg, and D. Hopwood, *Halo: Recursive proof composition without a trusted setup*, *IACR Cryptol. ePrint Arch.* **2019** (2019) 1021.
- [85] L. de Moura and N. Bjørner, *Z3: An efficient SMT solver*, in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.
- [86] Privacy & Scaling Explorations, “PSE zkEVM.” <https://github.com/privacy-scaling-explorations/zkevm-circuits>, 2024. Accessed: 2024-9-10.

- [87] A. Gabizon and Z. J. Williamson, “plookup: A simplified polynomial protocol for lookup tables.” Cryptology ePrint Archive, Paper 2020/315, 2020.
- [88] Privacy & Scaling Explorations, “Chiquito.” <https://github.com/privacy-scaling-explorations/chiquito>. Accessed: 2024-9-10.
- [89] Anoma, “Juvix.” <https://docs.juvix.org/>, 2021.
- [90] R. Joshi, G. Nelson, and K. Randall, *Denali: a goal-directed superoptimizer*, in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, (New York, NY, USA), p. 304–314, Association for Computing Machinery, 2002.
- [91] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, *Scaling up superoptimization*, in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), p. 297–310, Association for Computing Machinery, 2016.
- [92] Plonky3 Contributors, “Plonky3: A Toolkit for Polynomial IOPs (PIOps).” <https://github.com/Plonky3/Plonky3>, 2024. Accessed: 2024-11-15.
- [93] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, *Marlin: Preprocessing zkSNARKs with universal and updatable SRS*, in *Advances in Cryptology – EUROCRYPT 2020* (A. Canteaut and Y. Ishai, eds.), (Cham), pp. 738–768, Springer International Publishing, 2020.
- [94] 0xParC, “Big integer arithmetic and secp256k1 ECC operations in circom.” <https://github.com/0xPARC/circom-ecdsa>, 2023. Accessed: 2024-11-15.
- [95] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, *cvc5: A versatile and industrial-strength SMT solver*, in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Fisman and G. Rosu, eds.), (Cham), pp. 415–442, Springer International Publishing, 2022.
- [96] Privacy & Scaling Explorations, “halo2.” <https://github.com/privacy-ethereum/halo2>, 2024. Accessed: 2024-9-10.
- [97] StarkWare Industries, “Arithmetization I: How to Represent Computation as Algebra.” <https://medium.com/starkware/arithmetization-i-15c046390862>, 2020. Accessed: 2024-11-15.

- [98] F. H. Soureshjani, M. Hall-Andersen, M. Jahanara, J. Kam, J. Gorzny, and M. Ahmadvand, *Automated analysis of halo2 circuits*, in *Proceedings of the 21st International Workshop on Satisfiability Modulo Theories (SMT 2023) co-located with the 29th International Conference on Automated Deduction (CADE 2023), Rome, Italy, July, 5-6, 2023* (S. Graham-Lengrand and M. Preiner, eds.), CEUR Workshop Proceedings, pp. 3–17, CEUR-WS.org, 2023.
- [99] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, *Component-based synthesis for complex apis*, in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL '17*, (New York, NY, USA), p. 599–612, Association for Computing Machinery, 2017.
- [100] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, *Oracle-guided component-based program synthesis*, in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, (New York, NY, USA), p. 215–224, Association for Computing Machinery, 2010.
- [101] C. Wang, A. Cheung, and R. Bodik, *Synthesizing highly expressive sql queries from input-output examples*, in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, (New York, NY, USA), p. 452–466, Association for Computing Machinery, 2017.
- [102] D. Lee, W. Lee, H. Oh, and K. Yi, *Optimizing homomorphic evaluation circuits by program synthesis and term rewriting*, in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, (New York, NY, USA), p. 503–518, Association for Computing Machinery, 2020.
- [103] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, *Porcupine: a synthesizing compiler for vectorized homomorphic encryption*, in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021*, (New York, NY, USA), p. 375–389, Association for Computing Machinery, 2021.