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

JUNRUI LIU, University of California at Santa Barbara, USA
JIAXIN SONG, University of Illinois at Urbana-Champaign, USA
YANNING CHEN, University of Toronto, Canada
HANZHI LIU, University of California at Santa Barbara, USA
HONGBO WEN, University of California at Santa Barbara, USA
LUKE PEARSON, Polychain Capital, USA
YANJU CHEN, University of California at Santa Barbara, USA
YU FENG, University of California at Santa Barbara, USA

Zero-knowledge proof (ZKP) applications require translating high-level programs into arithmetic circuits—a process that demands both correctness and efficiency. While recent DSLs improve usability, they often yield suboptimal circuits, and hand-optimized implementations remain difficult to construct and verify. We present Tabby, a synthesis-aided compiler that automates the generation of high-performance ZK circuits from high-level code. Tabby introduces a domain-specific intermediate representation designed for symbolic reasoning and applies sketch-based program synthesis to derive optimized low-level implementations. By decomposing programs into reusable components and verifying semantic equivalence via SMT-based reasoning, Tabby ensures correctness while achieving substantial performance improvements. We evaluate Tabby on a suite of real-world ZKP applications and demonstrate significant reductions in proof generation time and circuit size against mainstream ZK compilers.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Automatic programming*; • **Security and privacy** → Privacy-preserving protocols.

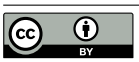
Additional Key Words and Phrases: Compilers, Program Synthesis, Zero-Knowledge Proofs

ACM Reference Format:
Junrui Liu, Jiaxin Song, Yanning Chen, Hanzhi Liu, Hongbo Wen, Luke Pearson, Yanju Chen, and Yu Feng. 2025. Tabby: A Synthesis-Aided Compiler for High-Performance Zero-Knowledge Proof Circuits. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 332 (October 2025), 27 pages. https://doi.org/10.1145/3763110

1 Introduction

Zero-knowledge proof (ZKP) systems [Goldwasser et al. 1985] have recently gained significant popularity among blockchain developers aiming to incorporate succinct verifiability and data privacy into their applications. These systems enable users to prove knowledge of a secret without revealing the secret itself, making them particularly valuable in blockchain contexts. Notably,

Authors' Contact Information: Junrui Liu, University of California at Santa Barbara, Santa Barbara, USA, junrui@cs.ucsb.edu; Jiaxin Song, University of Illinois at Urbana-Champaign, Santa Barbara, USA, jiaxins8@illinois.edu; Yanning Chen, University of Toronto, Toronto, Canada, yanning@cs.toronto.edu; Hanzhi Liu, University of California at Santa Barbara, Santa Barbara, USA, hanzhi@ucsb.edu; Hongbo Wen, University of California at Santa Barbara, Santa Barbara, USA, hongbowen@ucsb.edu; Luke Pearson, Polychain Capital, San Francisco, USA, luke@polychain.capital; Yanju Chen, University of California at Santa Barbara, Santa Barbara, USA, yanju@cs.ucsb.edu; Yu Feng, University of California at Santa Barbara, Santa Barbara, USA, yufeng@cs.ucsb.edu.



ACM 2475-1421/2025/10-ART332
https://doi.org/10.1145/3763110

privacy-centric digital currencies like ZCash [Ben Sasson et al. 2014] extensively utilize zero-knowledge proofs, and ZKP is also integral to layer-2 blockchain scaling solutions [Polygon 2022; Scroll 2022] by leveraging the succinct verifiability of ZKPs.

Technically, a zero-knowledge proof system involves two entities: a prover and a verifier. Given an input I selected by the verifier, the prover seeks to generate a proof demonstrating knowledge of a secret W that satisfies a specified relation $R(I, W)$. In most ZKP systems, this relation R is expressed through arithmetic circuits, which essentially represent polynomials over a finite field. Programming within the ZKP paradigm can be challenging for those lacking deep domain-specific knowledge of the underlying systems and the intricacies of writing zero-knowledge arithmetic circuits. Developers essentially need to adopt two mindsets: a cryptography-oriented perspective, knowledgeable about proving systems and foundational cryptographic principles; and an application developer's perspective, capable of intuitively handling public and private state to innovate and design novel blockchain applications. In practice, ZKP programmers must balance programmability and efficiency. Low-level frameworks and languages like Halo2 [Bowe et al. 2019] and Circom [Bellés-Muñoz et al. 2023] allow expert developers to perform fine-grained optimizations but are difficult to program and error-prone. High-level domain-specific languages (DSLs) such as Noir [Aztec 2022] and Leo [Chin 2021] offer user-friendliness with high-level programming constructs but often at the unacceptable expense of generating inefficient code, which can be up to 10 times slower [aayush thoughts 2023] than programming at the circuit level.

After extensive performance comparisons between circuits generated by mainstream compilers (e.g., Leo, Noir) and low-level circuits fine-tuned by domain experts, we identified the root cause of this performance gap. Existing compilers translate circuits from high-level DSLs to predefined and low-level polynomial constraints (i.e., *standard gates*) that encode the computation to be proven. This inefficiency arises because for complex operations—especially those involving loops with complex control flows—existing compilers natively unroll the loops, leading to a large number of statements and variables. This, in turn, requires a large number of standard gates, increasing the circuit size and consequently the proving time. To address this problem, domain experts bypass the compilers and design *custom gates* that define specialized polynomial constraints tailored to specific computational patterns within their circuits. By aligning the gate design with the specific arithmetic patterns of the computation, developers can exploit the inherent structures within the problem, reducing the number of gates and constraints needed.

However, designing custom gates on the fly and tailoring them for particular problems is non-trivial because developers need to (1) understand the original computation and (2) design custom gates that lead to equivalent computations while minimizing the number of polynomial constraints. One potential way to mitigate this problem is to leverage super-optimization [Joshi et al. 2002] to 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 [Phothilimthana et al. 2016], making them unsuitable for handling the complexity of ZKP circuits.

In this paper, we introduce Tabby, a compiler for generating high-performance ZKP circuits. 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, which are frequently used by upstream applications. Second, unlike existing ZKP compilers that only perform

standard compiler optimizations, Tabby leverages program synthesis [Feng et al. 2018; Solar-Lezama 2008; Torlak and Bodik 2014] to search for a new circuit that (a) is equivalent to the original one and (b) leverages custom gates to reduce the proving time. 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 [Aztec 2022; Bellés-Muñoz et al. 2023], 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 [Plonky3 Contributors 2024]), where it achieves significant performance gains in a case study, including up to 50× improvement in proving time.

In summary, this paper 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 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.

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 [Aztec 2022] to explain the Tabby workflow.

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 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 [Aztec 2022] 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 1 (A) shows a simple example in Noir [Aztec 2022]. 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 [Gabizon et al. 2019], Groth16 [Groth 2016],

and Marlin [Chiesa et al. 2020]. 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 [Benarroch et al. 2019] and PLONKish [Chen et al. 2023; Gabizon et al. 2019; Plonky3 Contributors 2024].

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 $\mathfrak{R}(\overrightarrow{z^n}, A^{nxk}, B^{nxk}, C^{nxk})$, 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: $\sum_{i\in[k]} A_i\overrightarrow{z^n} \cdot \sum_{i\in[k]} B_i\overrightarrow{z^n} - \sum_{i\in[k]} C_i\overrightarrow{z^n} = 0$. So at each step, we take the row i of the matrices A, B, C and multiply each row individually with $\overrightarrow{z^n}$. 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.

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

This now gives us the names of all variables we will use in the R1CS inputs and witness vectors. Thus, $\overrightarrow{z^n}$ will look like $[1, x, out, sym1, sym2, sym3]$ 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 $\overrightarrow{z^n}$, 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 $\overrightarrow{z^n}$.
(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 $\mathfrak{R}(\overrightarrow{z^n}, A^{nxk}, B^{nxk}, C^{nxk})$ as:

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

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

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

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 1. Particularly, given an array vals, the original circuit Ⓐ 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.

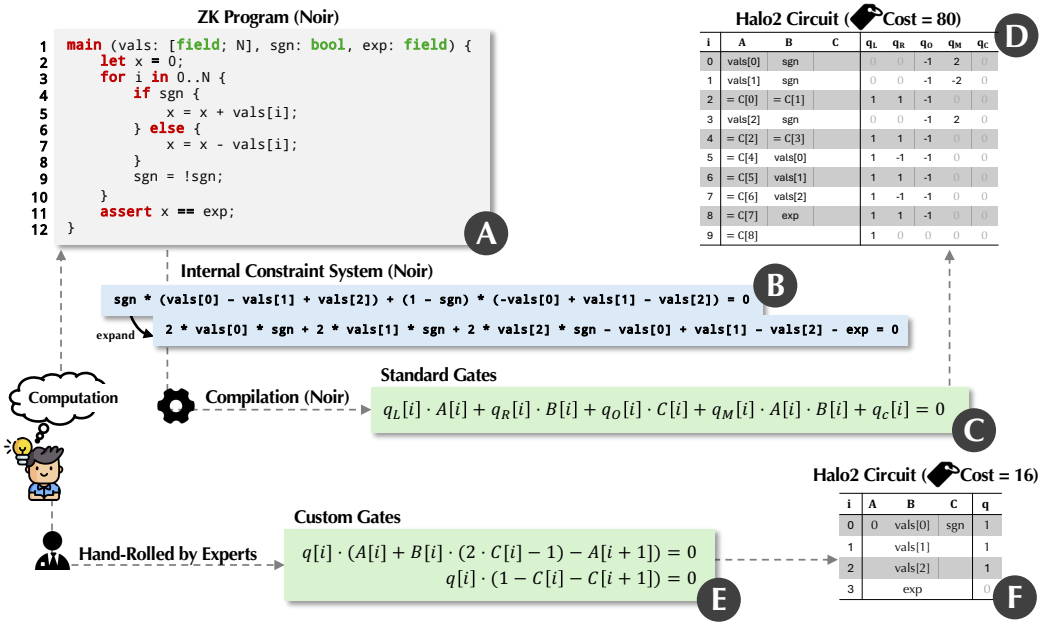


Fig. 1. Illustration showing how a circuit can be encoded as a trace table via standard gates for the Halo2 proving system ($A \longrightarrow B \longrightarrow C \longrightarrow D$). In order to optimize the size and performance of the circuit, experts write custom gates to dramatically shrink the size of the generated trace table ($A \longrightarrow E \longrightarrow F$), which requires non-trivial efforts.

In order for downstream cryptographic libraries to consume and process the circuit, prevailing ZK programming languages and compilers (such as Noir [Aztec 2022] and Circom [Bellés-Muñoz et al. 2023]) 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 Ⓑ). Figure 1 Ⓓ shows such an object — a table — that is used by one of the popular ZK proving systems called Halo2 [Bowe et al. 2019]. 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 1 Ⓒ:

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

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:

$$\begin{aligned} 0 \cdot \text{vals}[0] + 0 \cdot \text{sgn} + (-1) \cdot C[i] + 2 \cdot \text{vals}[0] \cdot \text{sgn} + 0 = 0, \\ \text{which evaluates to } \ 2 \cdot \text{vals}[0] \cdot \text{sgn} = C[i]. \end{aligned} \tag{1}$$

Equation 1 corresponds to the first term of the expanded version of the internal constraint system Ⓑ. When proving the correctness of the execution, one needs to submit the actual value of x during execution, which is then plugged into Equation 1. Only when the result checks out, for example:

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

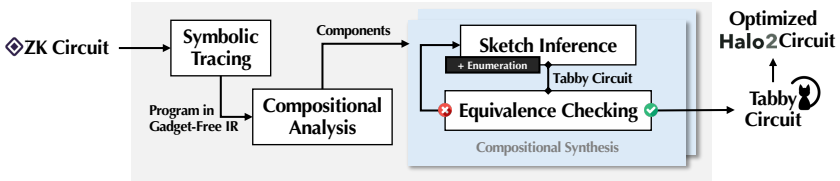


Fig. 2. An overview of Tabby.

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 1 Ⓓ, 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 1 Ⓓ 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 1 Ⓕ 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 1 Ⓔ. 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 [0xParC 2023].

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 [Scroll 2022], 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 a novel 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 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 preprocesses the original circuit into its internal representation (IR). Then Tabby runs a compositional analysis procedure that identifies different independent components of the circuit, where each component corresponds to an IR snippet. For each component, Tabby then invokes a sketched-based

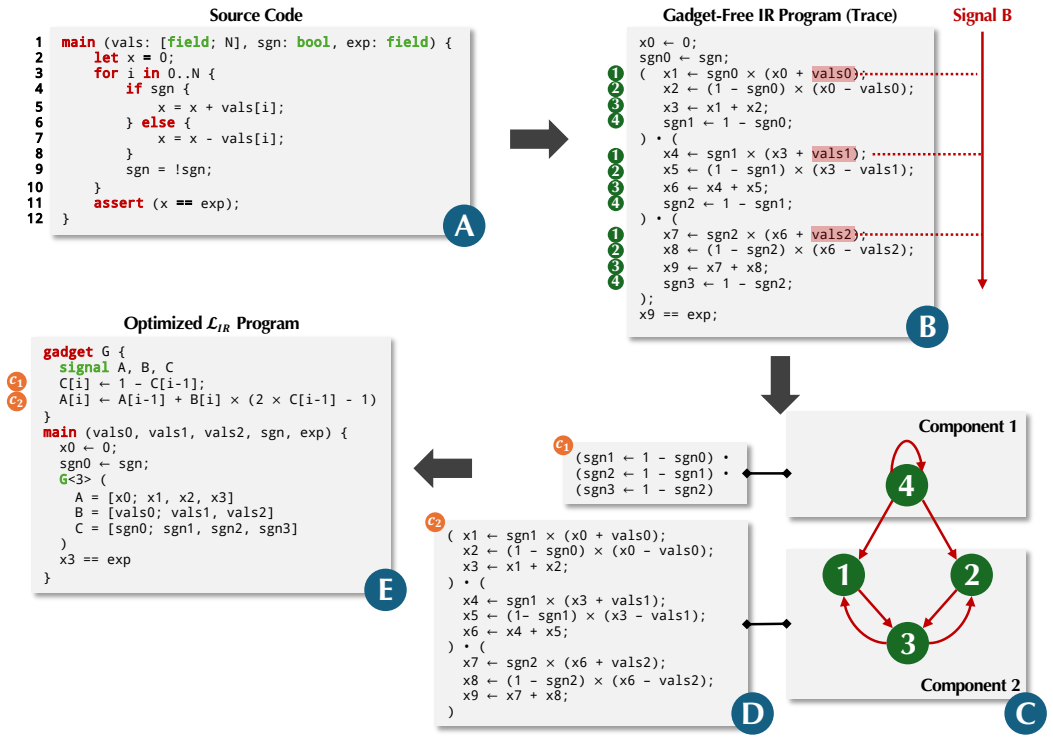


Fig. 3. A walk-through example showing how Tabby finds an optimized version of a given circuit.

program synthesis procedure to search for its equivalent counterpart written in $\mathcal{L}_{\mathsf{IR}}$ with optimized complexity. Finally, by assembly of the $\mathcal{L}_{\mathsf{IR}}$ 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 $\mathcal{L}_{\mathsf{IR}}$ in Figure 3, which breaks down to the following steps:

- Starting with the source code in Figure 3 (A), Tabby evaluates it into a program written in its intermediate representation, i.e., a trace program, as shown in Figure 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 3 (C). 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 3 (D).

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 $\{c\}$ else $\{c\}$	conditional
	|	assert $e = e$	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	::=	gadget G $\{S^*; s^*\}$	**Gadget**
σ	::=	$[x^*; x^*]$	**Signal literal**
δ	$\in$	$\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$	
n	$\in$	$\mathbb{N} = \{0, 1, 2, \ldots\}$	

Fig. 4. Syntax of $\mathcal{L}_{\mathsf{ZK}}$ (left) and syntax of the intermediate representation $\mathcal{L}_{\mathsf{IR}}$ (right). Constructs colored in blue can only appear in a gadget, while those in red cannot.

- Then for each component, Tabby invokes a sketch-based program synthesis algorithm that searches for its equivalent version written in the proposed $\mathcal{L}_{\mathsf{IR}}$ language. As shown in Figure 3 (E), Tabby introduces a language construct called *gadget*, which encapsulates the two synthesized custom gates (c_1) and (c_2) which correspond to the pattern observed in (D). The gadget G is expanded in the main program that produces corresponding constraints.
- Through Tabby's backend, the $\mathcal{L}_{\mathsf{IR}}$ 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.

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 $\mathcal{L}_{\mathsf{ZK}}$, a typical high-level DSL for ZKP (Section 3.1). We then give the syntax and the semantics of our novel IR, $\mathcal{L}_{\mathsf{IR}}$ (Section 3.2), and a set of translation rules from $\mathcal{L}_{\mathsf{ZK}}$ to $\mathcal{L}_{\mathsf{IR}}$. We finally give a precise definition of the synthesis problem in Section 3.4.

3.1 The High-Level ZK DSL $\mathcal{L}_{\mathsf{ZK}}$

The high-level source language that we consider here — which we shall denote by $\mathcal{L}_{\mathsf{ZK}}$— 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 [Aztec 2022; Bellés-Muñoz et al. 2023; Liu et al. 2024]. As shown in Figure 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 ×, 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

$$\frac{}{\iota, v \vdash f \Downarrow f}\ \text{G-Const} \qquad \frac{x = \sigma(\iota + \delta)}{\iota, v \vdash \sigma[i + \delta] \Downarrow v(x)}\ \text{G-Rot}$$

$$\frac{\iota, v \vdash e_1 \Downarrow f_1 \qquad \iota, v \vdash e_2 \Downarrow f_2 \qquad f_3 = f_1 \oplus f_2}{\iota, v \vdash e_1 \oplus e_2 \Downarrow f_3}\ \text{G-Binop} \qquad \frac{\iota, v \vdash e \Downarrow f \qquad x = \sigma(\iota + \delta)}{\iota, v \vdash (\sigma[i + \delta] \leftarrow e) \longrightarrow v[x \mapsto f]}\ \text{G-Assign}$$

$$\frac{\iota, v \vdash e_1 \Downarrow f_1 \qquad \iota, v \vdash e_2 \Downarrow f_2 \qquad f_1 = f_2}{\iota, v \vdash (e_1 = e_2) \longrightarrow v}\ \text{G-Constrain} \qquad \frac{}{\iota, v \vdash (\cdot) \longrightarrow v}\ \text{G-Step0}$$

$$\frac{\iota, v \vdash s \longrightarrow v' \qquad \iota, v' \vdash s' \longrightarrow v''}{\iota, v \vdash (s; s') \longrightarrow v''}\ \text{G-StepS} \qquad \frac{}{\iota, v \vdash G\langle 0 \rangle\ (\overrightarrow{S : \sigma}) \longrightarrow v}\ \text{G-Unroll0}$$

$$\frac{n > 0 \qquad G = \text{gadget } G\ \{\_; s\} \qquad \iota, v \vdash [\overrightarrow{\sigma/S}]s \longrightarrow v' \qquad \iota + 1, v' \vdash G\langle n - 1 \rangle\ (\overrightarrow{S = \sigma}) \longrightarrow v''}{\iota, v \vdash G\langle n \rangle\ (\overrightarrow{S = \sigma}) \longrightarrow v''}\ \text{G-UnrollS}$$

Fig. 5. Semantics of gadgets.

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 assert $e_1 = e_2$, 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.

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.

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 [Gabizon and Williamson 2020b] 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 (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

[1] A 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.

$$\frac{}{v \vdash \cdot \longrightarrow v}\ \text{M-Empty} \qquad \frac{v \vdash m \longrightarrow v' \qquad 0, v' \vdash s \longrightarrow v''}{v \vdash m; s \longrightarrow v''}\ \text{M-Stmt}$$

$$\frac{v \vdash m \longrightarrow v' \qquad 0, v' \vdash G\langle n\rangle(\overrightarrow{S = \sigma}) \longrightarrow v''}{v \vdash m; G\langle n\rangle(\overrightarrow{S = \sigma}) \longrightarrow v''}\ \text{M-Gadget}$$

$$\frac{v_{i-1} \vdash s_{1i} \longrightarrow v_i \text{ for } 1 \le i \le n \qquad \begin{array}{c} v \vdash m \longrightarrow v_0 \\ v_n \vdash (s_{21}; \ldots; s_{2n}) \bullet \cdots \bullet (s_{m1}; \ldots; s_{mn}) \longrightarrow v' \end{array}}{v \vdash m; (s_{11}; \ldots; s_{1n}) \bullet (s_{21}; \ldots; s_{2n}) \bullet \cdots \bullet (s_{m1}; \ldots; s_{mn}) \longrightarrow v'}\ \text{M-Trace}$$

Fig. 6. Semantics of $\mathcal{L}_{\text{IR}}$.

form $S[i+\delta] \leftarrow e$, 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 $e_1 = e_2$, 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 $[x_{-n}, \ldots, x_{-1}; x_0, \ldots, x_{m-1}]$. The first part $[x_{-n}, \ldots, x_{-1}]$ is the initial state of the signal, and the second part $[x_0, \ldots, x_{m-1}]$ 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 in given in Figure 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 rules defining this judgment is straightforward, and can be found in Figure 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 3.1. Figure 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

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

```
gadget Sum {
    signal A, B
    B[i] <- B[i-1] + A[i]
}
main (x0, x1, x2, x3, expected) {
    sum0 <- 0;
    Sum<4>(
        A = [x0, x1, x2, x3],
        B = [sum0; sum1, sum2, sum3, sum4]
    );
    sum4 == expected
}
```

Fig. 7. An IR program that computes the sum of four input variables using a gadget.

$$\mu_1 \Delta \mu_2 = \{x \mapsto l \mid x \mapsto l_1 \in \mu_1, x \mapsto l_2 \in \mu_2, l_1 \neq l_2, l \text{ fresh}\} \qquad \mu_1 \sqcup \mu_2(x) = \begin{cases} l & \text{if } x \mapsto l \in \mu_1 \Delta \mu_2 \\ \mu_1(x) & \text{otherwise} \end{cases}$$

$$\phi(\mu_1, \mu_2) = [l \leftarrow l_1 + l_2 \mid x \mapsto l \in \mu_1 \Delta \mu_2] \qquad \frac{}{\mu \vdash x \Rightarrow \mu(x)}\ \textsc{Tr-Read}$$

$$\frac{\mu(x) = v \qquad l \text{ fresh}}{\gamma, \mu \vdash (x \ = \ e) \dashv \mu[x \mapsto l] \rightsquigarrow (l \leftarrow \gamma \times v)}\ \textsc{Tr-Write}$$

$$\frac{\mu \vdash e_1 \Rightarrow v_1 \qquad \mu \vdash e_2 \Rightarrow v_2}{\gamma, \mu \vdash (\text{assert } e_1 \ = \ e_2) \dashv \mu' \rightsquigarrow (\gamma \times (v_1 - v_2) = 0)}\ \textsc{Tr-Cons}$$

$$\frac{\mu \vdash e_1 \Rightarrow 0 \qquad \gamma, \mu \vdash c_1 \dashv \mu' \rightsquigarrow p}{\gamma, \mu \vdash \text{if } e_1 \ \{c_1\} \text{ else } \{c_2\} \dashv \mu' \rightsquigarrow p}\ \textsc{Tr-IfTrue} \qquad \frac{\mu \vdash e_1 \Rightarrow f \neq 0 \qquad \gamma, \mu \vdash c_2 \dashv \mu' \rightsquigarrow p}{\gamma, \mu \vdash \text{if } e_1 \ \{c_1\} \text{ else } \{c_2\} \dashv \mu' \rightsquigarrow p}\ \textsc{Tr-IfFalse}$$

$$\frac{\begin{array}{c}\mu \vdash e_1 \Rightarrow v \\ \gamma \times v, \mu \vdash c_1 \dashv \mu_1 \rightsquigarrow p_1 \qquad (1-\gamma) \times v, \mu \vdash c_2 \dashv \mu_2 \rightsquigarrow p_2 \qquad \mu' = \mu_1 \sqcup \mu_2 \qquad p' = \phi(\mu_1, \mu_2)\end{array}}{\gamma, \mu \vdash \text{if } e_1 \ \{c_1\} \text{ else } \{c_2\} \dashv \mu' \rightsquigarrow (p_1; p_2; p')}\ \textsc{Tr-If}$$

$$\frac{\gamma, \mu \vdash c_1 \dashv \mu' \rightsquigarrow p_1 \qquad \gamma, \mu' \vdash c_2 \dashv \mu'' \rightsquigarrow p_2}{\gamma, \mu \vdash (c_1 \ ; \ c_2) \dashv \mu'' \rightsquigarrow (p_1; p_2)}\ \textsc{Tr-Seq} \qquad \frac{\mu \vdash e \Rightarrow 0}{\gamma, \mu \vdash \text{repeat } e \ \{c\} \dashv \mu \rightsquigarrow \epsilon}\ \textsc{Tr-RepeatSkip}$$

$$\frac{\mu \vdash e \Rightarrow f \neq 0 \qquad \gamma, \mu \vdash c \dashv \mu' \rightsquigarrow p \qquad \gamma, \mu' \vdash \text{repeat } (f-1) \ \{c\} \dashv \mu'' \rightsquigarrow p'}{\gamma, \mu \vdash \text{repeat } e \ \{c\} \dashv \mu'' \rightsquigarrow (p \bullet p')}\ \textsc{Tr-RepeatStep}$$

Fig. 8. Selected translation rules from $\mathcal{L}_{\text{ZK}}$ to $\mathcal{L}_{\text{IR}}$.

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.

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 8 [Ozdemir et al. 2022]. These rules transform

high-level constructs such as variable assignments, constraints, conditionals, 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 $\mathcal{L}_{\mathsf{ZK}}$ variables to $\mathcal{L}_{\mathsf{IR}}$ symbolic locations, and field expression γ to represent the current path condition. We use the judgment $\gamma, \mu \vdash s \dashv \mu' \rightsquigarrow p$ 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 $\bullet$.

3.4 Problem Definition

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

Definition 3.2. **(Partial program)** Given a $\mathcal{L}_{\mathsf{IR}}$ 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 3.3. **(Gadget cost function Θ)** The gadget cost function $\Theta : g \to \mathbb{N}$ 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 $\mathcal{L}_{\mathsf{IR}}$ 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:

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

Algorithm 1 Synthesis algorithm.

```
1:  procedure Synthesis(p)
2:      Input: Initial IR program p lowered from source program in L_ZK
3:      Output: Optimized IR program p_⋄
4:      C, p' ← Decompose(p)          ▷ Decompose p into components C and partial program p'
5:      G ← ∅                                                    ▷ Initialize solutions
6:      E ← GenRandomExamples()                                  ▷ Initialize examples
7:      for each c ∈ C do                                        ▷ Iterate over each component p
8:          g* ← InferSketch(c)                                  ▷ Infer a symbolic gadget sketch
9:          while g ← Enumerate(g*) do                           ▷ Enumerate feasible gadgets from sketch
10:             if Equiv(g, c) then                              ▷ Verify gadget is equivalent to component
11:                 G[c] ← g                                     ▷ Add synthesized gadget g to solutions
12:                 break
13:             else
14:                 E ← Cex(g, c)                                ▷ Learn counterexample
15:             end if
16:         end while
17:     end for
18:     p_⋄ ← Assemble(G, p')          ▷ Assemble synthesized G and partial program p'
19:     return p_⋄
20: end procedure
```

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 3.4. **($\mathcal{L}_{\text{IR}}$ 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 3.5. **(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 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.

4.1 Overview

The synthesis algorithm is shown in Algorithm 1. At a high level, our synthesis procedure follows the well-established Counterexample-Guided Inductive Synthesis (CEGIS) paradigm [Feng et al. 2018; Solar-Lezama 2008]. 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 1 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 4.2) 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 4.3) 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). 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 $p_\diamond$.

4.2 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:

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

where m is the number of iterations in the loop trace t_i, and n is the number of statements in each iteration[3]. We refer to each $(s_{j1}; \ldots; s_{jn})$ 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 $\{t_1, \ldots, t_k\}$ 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.

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

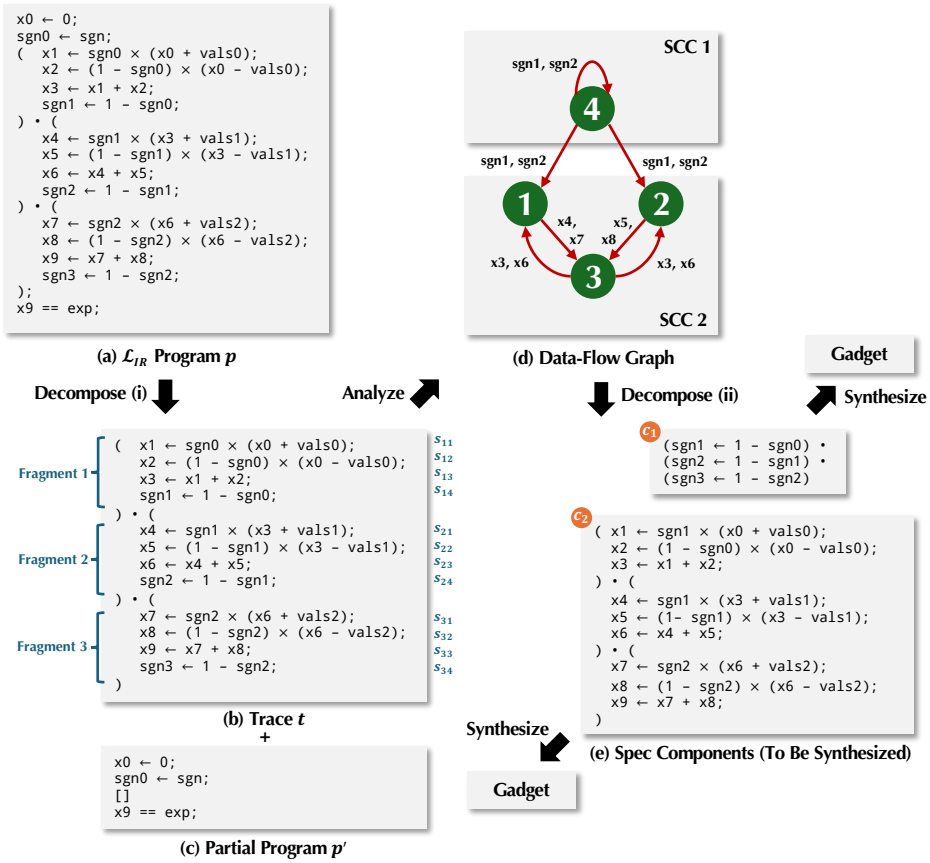


Fig. 9. Illustration of an example decomposition procedure.

The second purpose of decomposition is to make synthesis scalable by further breaking down each synthesis problem t_i into a sequence of *components* $C = [c_1, c_2, \ldots, c_L]$, 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 $c_1, \ldots, c_L$ 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 $V = \{1, \ldots, n\}$ 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 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 4.1. Consider the $\mathcal{L}_{\text{IR}}$ trace program p shown in Figure 9, which is generated from the $\mathcal{L}_{\text{ZK}}$ program Figure 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 shwon 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

$$\frac{}{f \Rightarrow_R \varnothing}\ \textsc{Read-Const} \qquad \frac{\sigma^{-1}(x) = s}{x \Rightarrow_R \{s\}}\ \textsc{Read-Var} \qquad \frac{e_1 \Rightarrow_R S_1 \qquad e_2 \Rightarrow_R S_2}{e_1 \oplus e_2 \Rightarrow_R S_1 \cup S_2}\ \textsc{Read-Binop}$$

$$\frac{e \Rightarrow_R S}{(x \leftarrow e) \Rightarrow_R S}\ \textsc{Read-Assign} \qquad \frac{e_1 \Rightarrow_R S_1 \qquad e_2 \Rightarrow_R S_2}{(e_1 = e_2) \Rightarrow_R S_1 \cup S_2}\ \textsc{Read-Constrain}$$

$$\frac{\sigma^{-1}(x) = s}{(x \leftarrow e) \Rightarrow_W \{s\}}\ \textsc{Write-Assign} \qquad \frac{}{(e_1 = e_2) \Rightarrow_W \varnothing}\ \textsc{Write-Constrain}$$

$$\frac{}{\cdot \Rightarrow_\xi \varnothing}\ \textsc{Step0} \qquad \frac{s_1 \Rightarrow_\xi S_1 \qquad s_2 \Rightarrow_\xi S_2}{(s_1; s_2) \Rightarrow_\xi S_1 \cup S_2}\ \textsc{StepS}$$

Fig. 10. Sketch inference rules.

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.

4.3 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 4.2. **(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 10.

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

Example 4.3. Consider the specification component c_2 from Figure 9. Suppose we identify from the loop trace the signal $s = [\texttt{vals0}, \texttt{vals1}, \texttt{vals2}]$. 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 Equivalence Checking

Once a gadget g is enumerated by Algorithm 1, a procedure Equiv(g, c) is then invoked to verify the equivalence between the proposed gadget g and the specification component c. Given the

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

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 ([Bellés-Muñoz et al. 2023]).

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 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 [Barbosa et al. 2022] equipped with the theory of finite fields [Ozdemir et al. 2023].

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 [Bellés-Muñoz et al. 2023].

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.

6 Evaluation

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

Table 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		sort	205
Circom		Num2Bits	4,104
		MultiMux1	12,292
Circom	Utils	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

- **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 [Aztec 2022], SP1 [Succinct Labs [n. d.]] and Circom [Bellés-Muñoz et al. 2023]), 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 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.

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 6.1 and Section 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 [Privacy & Scaling Explorations 2024a].[5] The

[5]The 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 2. A comparison of circuit performance between those optimized by Tabby and Noir.

Benchmark	Compile Time (s)		Proof Size (bytes)		Proving Time (s)		Table Size (# of cells)	
	Noir	Tabby	Noir	Tabby	Noir	Tabby	Noir	Tabby
fib	1.03	**0.43**	1,760	**384** (-78.2%)	0.16	**0.05** (-68.8%)	450	**125** (-72.2%)
check_for_collision	37.58	**25.82**	1,760	**608** (-65.4%)	29.0	**0.48** (-98.3%)	1,247,220	**223,660** (-82.1%)
check_ship_range	**1.21**	2.05	1,760	**544** (-69.1%)	1.04	**0.71** (-31.7%)	12,600	**8,365** (-33.6%)
has_ship	37.10	**11.91**	1,760	**512** (-70.9%)	28.51	**0.66** (-97.7%)	1,474,740	**102,440** (-93.1%)
check_hit	1.41	**0.22**	1,760	**416** (-76.4%)	0.60	**0.05** (-91.7%)	13,770	**11,244** (-18.3%)
dotproduct	17.62	**6.94**	1,760	**448** (-74.5%)	16.15	**3.52** (-78.2%)	180,180	**61,470** (-65.9%)
check_cards	1.70	**1.38**	1760	**544** (-69.1%)	29.30	**0.23** (-99.2%)	1,888,470	**24,240** (-98.7%)
pow	**1.98**	2.20	1,760	**512** (-70.9%)	1.54	**0.49** (-68.2%)	31,950	**5,115** (-84.0%)
sqrt	3.27	**2.84**	1,760	**640** (-63.6%)	2.91	**0.12** (-95.9%)	78,210	**31,626** (-59.6%)
sort	**1.45**	3.74	1,760	**512** (-70.9%)	0.60	**0.23** (-61.7%)	9,180	**2,540** (-72.3%)
Averaged	10.43	**5.75**	1,760	**512** (-70.9%)	10.98	**0.65** (-94.0%)	493,677	**47,083** (-90.5%)
Median	**1.84**	2.52	1,760	**512** (-70.9%)	2.22	**0.36** (-84.0%)	55,080	**17,742** (-67.8%)

Table 3. A comparison of circuit performance between those optimized by Tabby and CirC.

Benchmark	Compile Time (s)		Proof Size (bytes)		Proving Time (s)		Table Size (# of cells)	
	CirC	Tabby	CirC	Tabby	CirC	Tabby	CirC	Tabby
fib	**0.06**	0.43	1,760	**384** (-78.2%)	0.14	**0.05** (-64.3%)	480	**125** (-74.0%)
check_for_collision	**2.29**	25.82	1,760	**608** (-65.5%)	52.06	**5.95** (-88.6%)	3,932,160	**223,660** (-94.3%)
check_ship_range	**1.94**	2.05	1,760	**544** (-69.1%)	48.93	**0.71** (-98.5%)	3,932,160	**8,365** (-99.8%)
has_ship	**2.70**	11.91	1,760	**512** (-70.9%)	46.27	**0.66** (-98.6%)	3,932,160	**102,440** (-97.4%)
check_hit	**0.05**	0.22	1,760	**416** (-76.4%)	0.33	**0.05** (-84.8%)	**7,680**	11,244 (+46.4%)
dotproduct	**0.56**	6.94	1,760	**448** (-74.5%)	4.09	**3.52** (-13.9%)	245,760	**61,470** (-75.0%)
check_cards	**0.06**	1.38	1760	**544** (-69.1%)	4.13	**0.23** (-94.4%)	245,760	**24,240** (-90.1%)
sort	**0.02**	3.74	1,760	**512** (-70.9%)	0.51	**0.23** (-54.9%)	15,360	**2,540** (-83.5%)
Averaged	**0.96**	6.56	1,760	**496** (-71.8%)	19.56	**1.43** (-92.7%)	1,538,940	**54,260** (-96.5%)
Median	**0.31**	2.90	1,760	**512** (-70.9%)	4.11	**0.45** (-89.2%)	245,760	**17,742** (-92.8%)

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 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.

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 [Aztec 2022], Circom [Bellés-Muñoz et al. 2023] and CirC [Ozdemir et al. 2022], the state-of-the-art ZK programming languages and compilers. We show the evaluation results in Table 2, Table 3 and Table 4.

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 (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, Tabby is capable of synthesizing custom gates on a per-circuit basis, which helps it reduce

Table 4. A comparison of circuit performance between those optimized by Tabby and Circom.

Benchmark	Compile Time (s)		Proof Size (bytes)		Proving Time (s)		Table Size (# of cells)	
	Circom	Tabby	Circom	Tabby	Circom	Tabby	Circom	Tabby
Decoder	**0.11**	2.85	1,760	**544** (-69.1%)	1.06	**0.38** (-64.2%)	30,720	**3,918** (-87.2%)
Num2bits	**6.21**	22.71	1,760	**480** (-72.7%)	17.41	**0.59** (-96.6%)	983,040	**81,970** (-91.7%)
Multimux1	**1.02**	27.28	1,760	**416** (-76.4%)	30.34	**4.95** (-83.7%)	1,966,080	**81,935** (-95.8%)
BigIsEqual	**0.62**	3.31	1,760	**576** (-67.3%)	17.01	**2.40** (-85.9%)	983,040	**30,126** (-96.9%)
MultiAND	**1.07**	2.51	1,760	**416** (-76.4%)	15.30	**3.33** (-78.2%)	491,520	**40,985** (-91.7%)
BigLessThan	**1.84**	7.63	1,760	**544** (-69.1%)	7.91	**0.29** (-96.3%)	491,520	**79,366** (-83.9%)
BigAddNoCarry	**1.73**	8.12	1,760	**544** (-69.1%)	22.84	**1.03** (-95.5%)	1,966,080	**241,626** (-87.7%)
BigSub	**1.75**	17.41	1,760	**544** (-69.1%)	38.17	**1.83** (-95.2%)	1,966,080	**473,298** (-75.9%)
BigAdd	**1.71**	14.02	1,760	**544** (-69.1%)	25.36	**0.86** (-96.6%)	1,966,080	**241,605** (-87.7%)
BigModSumThree	**0.12**	7.34	1,760	**544** (-69.1%)	1.35	**0.27** (-80.0%)	61,440	**57,568** (-6.30%)
BigModSumFour	**0.10**	7.86	1,760	**544** (-69.1%)	1.32	**0.29** (-78.0%)	61,440	**57,582** (-6.28%)
ModProd	**0.11**	0.44	1,760	**544** (-69.1%)	2.22	**0.03** (-98.6%)	122,880	**3,731** (-96.9%)
BigSubModP	**0.26**	29.94	1,760	**576** (-67.3%)	98.25	**1.65** (-98.3%)	7,864,320	**819,568** (-89.6%)
Averaged	**1.36**	10.12	1,760	**520** (-70.5%)	15.02	**1.35** (-91.0%)	1,458,018	**170,252** (-87.4%)
Median	**1.04**	7.99	1,760	**544** (-69.1%)	17.01	**0.86** (-94.9%)	983,040	**79,366** (-91.8%)

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, Tabby is still able to reduce the proof size by more than 60%. In the comparison of the actual proving time for each compiled circuit, Tabby 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 Tabby consistently achieves comparable compilation latency compared to Noir, Circ, and Circom. As shown in the **Compile Time** column of all tables, Tabby 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 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.

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 4.3) and *decomposition* (Section 4.2); to do this, we create the following ablative versions of Tabby:

- **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 5, 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× faster than the baseline version (respectively, 54× than the +S version and 18× than the +D version). A closer look at the benchmarks solved indicates that both the sketch inference and decomposition actually help scale

Table 5. 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
fib	0.01	0.01	0.01	0.01
check_for_collision	-	-	26.92	4.39
check_ship_ranges	-	-	1.55	0.11
has_ship	-	-	-	12.51
check_hit	0.01	0.01	0.01	0.01
dotproduct	6.02	6.02	6.2	6.04
check_cards	0.11	0.10	0.11	0.10
pow	-	-	1.98	1.93
sqrt	-	-	2.01	1.98

Benchmark	Synthesis Time (s)			
	Base.	+S	+D	Full
sort	-	-	7.82	7.17
Decoder	-	-	0.26	0.07
Num2Bits	-	-	4.00	0.35
MultiMux1	0.93	0.95	0.93	0.93
BigIsEqual	-	-	0.21	0.06
multiAND	1.04	1.03	1.04	1.05
BigLessThan	-	-	8.68	0.82
BigAddNoCarry	-	-	-	11.47
BigSub	-	-	-	11.79
Average	>400	>400	>136	7.32

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 synthesis time.

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

6.3 A Case Study of Tabby for Optimizing Other Languages (RQ3)

Although we target PLONKish circuits in the previous experiments, with additional engineering effort, Tabby 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 Tabby can be applied to synthesize optimized AIR [StarkWare Industries 2020] circuits, another popular arithmetization behind zkSTARKs [Ben-Sasson et al. 2018]. 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.

We use the Fibonacci computation as our case study. The source program in $\mathcal{L}_{\text{ZK}}$ is shown in Figure 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 [Plonky3 Contributors 2024]), is shown in Figure 11 (c). The corresponding AIR table filled with witness values is shown in Figure 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 Plonky3[6] results in a proving time of 30.5 seconds. For comparison, we compile a similar Fibonacci program using SP1 [Succinct Labs [n. d.]], a ZK compiler implemented in Plonky3 that supports Rust as 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.

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

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

```
main () {
  let x = 0;
  let y = 1;
  for _ in 0..5 {
    let z = x + y;
    x = y;
    y = z;
  }
  assert (x == 5);
}
```

(a)

```
gadget F {
  signal A, B
  A[0] ← B[-1];
  B[0] ← A[-1] + B[-1]
}
main () {
  x0 ← 0;
  y1 ← 1;
  F<5> (
    A = [x0; x1, x2, x3, x4, x5],
    B = [y0; y1, y2, y3, y4, y5]
  )
}
```

(b)

```
when first_row {
  assert curr.A == 0;
  assert curr.B == 1;
}
when transition {
  assert next.A == curr.B;
  assert next.B == curr.A + curr.B;
}
when last_row {
  assert curr.A == 5;
}
```

(c)

A	B
0	1
1	1
1	2
2	3
3	5
5	8

(d)

Fig. 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.

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

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 [Ben-Sasson et al. 2014] 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 [Groth 2016] and PLONK [Gabizon et al. 2019], 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 [Chen et al. 2023], TurboPLONK [Gabizon and Williamson 2020b] and UltraPLONK [Williamson 2021]) share the same PLONK style arithmetization, with compatible extensions, which are collectively referred to as PLONKish arithmetization.

zk-STARK [Ben-Sasson et al. 2018] is one of the other popular ZK proof systems, whose typical realizations such as Cairo [Goldberg et al. 2021] base themselves on AIR[8] 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 [Bellés-Muñoz et al. 2023], ZoKrates [Eberhardt and Tai 2018], and Noir [Aztec [n. d.]] 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 their computation. Other languages like Leo [Chin 2021], Cairo [Goldberg et al. 2021] and zkEVM [Privacy & Scaling Explorations 2024b; Scroll [n. d.]] 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

[8]AIR is short for *Algebraic Intermediate Representation*.

built upon Halo2 [Zcash [n. d.]], 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 [RISC Zero [n. d.]], SP1 [Succinct Labs [n. d.]] and Jolt [Arun et al. 2023] "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 [Bellés-Muñoz et al. 2023] compiles its programs into R1CS, and performs constraint-level optimization [Albert et al. 2022]. For PLONKish arithmetization, [Ambrona et al. 2022] uses a predefined set of rewrite rules to manually optimize TurboPLONK circuits, while the static analysis framework proposed in [Soureshjani et al. 2023] to detect common circuit security vulnerabilities may be adapted to perform simple optimizations. Leo [Chin 2021] 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 [Gabizon and Williamson 2020a] and CirC [Ozdemir et al. 2022].

There are also dedicated IRs that enables compiler optimization towards different goals. For example, Noir [Aztec [n. d.]], Chiquito [Privacy & Scaling Explorations [n. d.]] and Juvix [Anoma 2021] are designed for better usability as well as compatibility with specific blockchains. Coda [Liu et al. 2024] is designed for composing safe ZK circuits. CirC [Ozdemir et al. 2022] focuses more on compatibility between different ZK langauges. 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 [Solar-Lezama 2008; Torlak and Bodik 2014] and component-based synthesis [Feng et al. 2018, 2017a,b; Jha et al. 2010]. Sketch [Solar-Lezama 2008] 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 [Feng et al. 2018, 2017a,b; Wang et al. 2017], which construct complex programs by composing reusable subcomponents, often guided by lightweight pruning techniques based on abstract semantics. Systems such as Neo [Feng et al. 2018] and Morpheus [Feng et al. 2017a] 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 [Cowan et al. 2021; Lee et al. 2020]. 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 [Lee et al. 2020] 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 [Ozdemir et al. 2023]. Despite these differences, prior work in this area largely relies on established program synthesis techniques such as rewrite-based optimization [Lee et al. 2020] and sketch-based synthesis [Cowan et al. 2021], the latter of which is closely related to the approach adopted by Tabby.

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.

Data-Availability Statement

The source code, benchmarks, and scripts utilized in the experiments are available as an accompanying artifact [Liu et al. 2025].

Acknowledgments

This work is supported in part by Google Faculty Research Award, Ethereum Foundation Academic Award, NSF 1908494, and DARPA N66001-22-2-4037.

References

0xParC. 2023. Big integer arithmetic and secp256k1 ECC operations in circom. https://github.com/0xPARC/circom-ecdsa. Accessed: 2024-11-15.

aayush thoughts. 2023. An Opinionated Overview of ZK Tooling and Proof Systems Right Now. https://blog.aayushg.com/zk/. Accessed: 2024-11-15.

Elvira Albert, Marta Bellés-Muñoz, Miguel Isabel, Clara Rodríguez-Núñez, and Albert Rubio. 2022. Distilling Constraints in Zero-Knowledge Protocols. In *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I* (Haifa, Israel). Springer-Verlag, Berlin, Heidelberg, 430–443. doi:10.1007/978-3-031-13185-1_21

Miguel Ambrona, Anne-Laure Schmitt, Raphael R. Toledo, and Danny Willems. 2022. New optimization techniques for PlonK's arithmetization. Cryptology ePrint Archive, Paper 2022/462. https://eprint.iacr.org/2022/462

Anoma. 2021. Juvix. https://docs.juvix.org/.

Arasu Arun, Srinath Setty, and Justin Thaler. 2023. Jolt: SNARKs for Virtual Machines via Lookups. Cryptology ePrint Archive, Paper 2023/1217. https://eprint.iacr.org/2023/1217

Aztec. [n. d.]. Noir. https://github.com/noir-lang/noir. Accessed: 2024-9-10.

Aztec. 2022. Introducing Noir. https://noir-lang.org/.

Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442. doi:10.1007/978-3-030-99524-9_24

Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2023. Circom: A Circuit Description Language for Building Zero-Knowledge Applications. *IEEE Transactions on Dependable and Secure Computing* 20, 6 (2023), 4733–4751. doi:10.1109/TDSC.2022.3232813

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

Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. 459–474. doi:10.1109/SP.2014.36

Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 781–796. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson

Daniel Benarroch, Kobi Gurkan, Ron Kahat, Aurélien Nicolas, and Eran Tromer. 2019. zkInterface, a standard tool for zero-knowledge interoperability. https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-zkinterface.pdf.

Sean Bowe, Jack Grigg, and Daira Hopwood. 2019. Recursive Proof Composition without a Trusted Setup. Cryptology ePrint Archive, Paper 2019/1021. https://eprint.iacr.org/2019/1021

Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. 2023. 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* (Lyon, France). Springer-Verlag, Berlin, Heidelberg, 499–530. doi:10.1007/978-3-031-30617-4_17

Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Advances in Cryptology – EUROCRYPT 2020*, Anne Canteaut and Yuval Ishai (Eds.). Springer International Publishing, Cham, 738–768. doi:10.1007/978-3-030-45721-1_26

Collin Chin. 2021. LEO: A Programming Language for Formally Verified, Zero-Knowledge Applications. https://docs.zkproof.org/pages/standards/accepted-workshop4/proposal-leo.pdf.

Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. 2021. Porcupine: a synthesizing compiler for vectorized homomorphic encryption. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 375–389. doi:10.1145/3453483.3454050

Jacob Eberhardt and Stefan Tai. 2018. 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)*. 1084–1091. doi:10.1109/Cybermatics_2018.2018.00199

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 420–435. doi:10.1145/3192366.3192382

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017a. 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* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 422–436. doi:10.1145/3062341.3062351

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017b. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 599–612. doi:10.1145/3009837.3009851

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

Ariel Gabizon and Zachary J. Williamson. 2020b. The turbo-plonk program syntax for specifying snark programs. In *Proceedings of the 3rd ZKProof Workshop*.

Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Paper 2019/953. https://eprint.iacr.org/2019/953

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

Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1985. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing* (Providence, Rhode Island, USA) *(STOC '85)*. Association for Computing Machinery, New York, NY, USA, 291–304. doi:10.1145/22145.22178

Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 305–326. doi:10.1007/978-3-662-49896-5_11

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. doi:10.1145/1806799.1806833

Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 304–314. doi:10.1145/512529.512566

DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2020. Optimizing homomorphic evaluation circuits by program synthesis and term rewriting. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 503–518. doi:10.1145/3385412.3385996

Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig, and Yu Feng. 2024. Certifying Zero-Knowledge Circuits with Refinement Types. In *2024 IEEE Symposium on Security and Privacy (SP)*. 1741–1759. doi:10.1109/SP54263.2024.00078

Junrui Liu, Jiaxin Song, Yanning Chen, Hanzhi Liu, Hongbo Wen, Luke Pearson, Yanju Chen, and Yu Feng. 2025. Tabby: A Synthesis-aided Compiler for High-performance Zero-knowledge Proof Circuits. Zenodo. doi:10.5281/zenodo.16920330

Alex Ozdemir, Fraser Brown, and Riad S. Wahby. 2022. CirC: Compiler infrastructure for proof systems, software verification, and more. In *2022 IEEE Symposium on Security and Privacy (SP)*. 2248–2266. doi:10.1109/SP46214.2022.9833782

Alex Ozdemir, Gereon Kremer, Cesare Tinelli, and Clark W. Barrett. 2023. Satisfiability Modulo Finite Fields. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13965)*, Constantin Enea and Akash Lal (Eds.). Springer, 163–186. doi:10.1007/978-3-031-37703-7_8

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 297–310. doi:10.1145/2872362.2872387

Plonky3 Contributors. 2024. Plonky3: A Toolkit for Polynomial IOPs (PIOPs). https://github.com/Plonky3/Plonky3. Accessed: 2024-11-15.

Polygon. 2022. Bring Ethereum to everyone. https://polygon.technology/polygon-zkevm.

Privacy & Scaling Explorations. [n. d.]. Chiquito. https://github.com/privacy-scaling-explorations/chiquito. Accessed: 2024-9-10.

Privacy & Scaling Explorations. 2024a. halo2. https://github.com/privacy-ethereum/halo2. Accessed: 2024-9-10.

Privacy & Scaling Explorations. 2024b. PSE zkEVM. https://github.com/privacy-scaling-explorations/zkevm-circuits. Accessed: 2024-9-10.

RISC Zero. [n. d.]. RISC Zero. https://github.com/risc0/risc0. Accessed: 2024-9-10.

Scroll. [n. d.]. Scroll zkEVM. https://github.com/scroll-tech/zkevm-circuits. Accessed: 2024-9-10.

Scroll. 2022. The Native zkEVM Scaling Solution for Ethereum. https://scroll.io/.

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

Fatemeh Heidari Soureshjani, Mathias Hall-Andersen, MohammadMahdi Jahanara, Jeffrey Kam, Jan Gorzny, and Mohsen Ahmadvand. 2023. Automated Analysis of Halo2 Circuits. Cryptology ePrint Archive, Paper 2023/1051. https://eprint.iacr.org/2023/1051

StarkWare Industries. 2020. Arithmetization I: How to Represent Computation as Algebra. https://medium.com/starkware/arithmetization-i-15c046390862. Accessed: 2024-11-15.

Succinct Labs. [n. d.]. SP1. https://github.com/succinctlabs/sp1. Accessed: 2024-9-10.

Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 530–541. doi:10.1145/2594291.2594340

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 452–466. doi:10.1145/3062341.3062365

Zac Williamson. 2021. Aztec's ZK-ZK-Rollup, looking behind the cryptocurtain. https://medium.com/aztec-protocol/aztecs-zk-zk-rollup-looking-behind-the-cryptocurtain-2b8af1fca619. Accessed: 2024-9-10.

Zcash. [n. d.]. halo2. https://github.com/zcash/halo2. Accessed: 2024-9-10.

Received 2025-03-26; accepted 2025-08-12