Zinnia: An Expressive and Efficient Tensor-Oriented Zero-Knowledge Programming Framework

ZHANTONG XUE, Hong Kong University of Science and Technology, China
PINGCHUAN MA*, Hong Kong University of Science and Technology, China and CipherInsight Limited, China
ZHAOYU WANG, Hong Kong University of Science and Technology, China
SHUAI WANG*, Hong Kong University of Science and Technology, China and CipherInsight Limited, China

Zero-knowledge proofs (ZKPs) are cryptographic protocols that enable a prover to convince a verifier of a statement's truth without revealing any details beyond its validity. Typically, the statement is encoded as an arithmetic circuit, and allows the prover to demonstrate that the circuit evaluates to true without revealing its inputs. Despite their potential to enhance privacy and security, ZKPs are difficult to write and optimize, limiting their adoption in machine learning and data science. To address these challenges, we introduce Zinnia, a zero-knowledge programming framework with high *utility*, *expressiveness* and *efficiency* for *tensor-oriented computation*. Zinnia provides a high-level programming language that enables developers to easily write ZKP programs, and it employs a novel symbolic execution-inspired approach to extracting semantics from these programs to generate arithmetic circuits. Zinnia supports tensor-oriented computations and provides a rich set of programming constructs, optimizations, and a powerful static type system for expressing and optimizing complex logic. We evaluate Zinnia across 25 real-world programming tasks and a user study, comparing it to existing solutions, including DSLs and zkVMs (Halo2, SP1, and RISC0). Our results demonstrate that Zinnia outperforms these baselines in utility, expressiveness, and efficiency, with a *statistically significant* reduction in development time, $2-3\times$ shorter code length, 19.3% smaller circuit size, and up to $245\times$ faster proving time compared to zkVMs, paving the way for practical ZKP applications in various domains.

1 INTRODUCTION

Zero-knowledge proofs (ZKPs) are cryptographic protocols enabling a prover to convince a verifier of a statement's truth without disclosing any details about the statement itself [8]. Typically, the statement is encoded as an arithmetic circuit, and allows the prover to demonstrate that the circuit evaluates to true without revealing its inputs. For example, in cryptocurrencies, ZKP has been used to enhance blockchain efficiency to reduce on-chain smart contract verification costs and improve throughput by converting Solidity programs into equivalent arithmetic circuits [10] (e.g., zkEVM [30]).[1] Rather than executing the contract, a blockchain node verifies the circuit-generated proof which shows that the contract would have executed correctly and is way more efficient than executing the contract itself.

For most ZKP systems, the process involves generating a proof that an arithmetic circuit C evaluates to true for a given input i and witness w. The prover generates a proof π, and the verifier checks its validity. Upon verification, the verifier is convinced that the prover knows a witness w such that $C(i, w)$ *is satisfiable*. Typically, C is a set of polynomial equations over finite fields. However, two major challenges remain: first, arithmetic circuits are difficult to write and debug

*Corresponding author
[1] In the context of this paper, unless otherwise specified, we use the term ZKP to refer to zk-SNARKs.

Authors' addresses: Zhantong Xue, Hong Kong University of Science and Technology, Hong Kong, China, zxueai@cse.ust.hk; Pingchuan Ma, Hong Kong University of Science and Technology, Hong Kong, China and CipherInsight Limited, Hong Kong, China, pmaab@cse.ust.hk; Zhaoyu Wang, Hong Kong University of Science and Technology, Hong Kong, China, zwangjz@cse.ust.hk; Shuai Wang, Hong Kong University of Science and Technology, Hong Kong, China and CipherInsight Limited, Hong Kong, China, shuaiw@cse.ust.hk.

compared to high-level programming languages, and second, optimizing ZKP performance demands specialized expertise in cryptography and circuit design.

To simplify arithmetic circuit development, various ZKP languages and compilers have been introduced. These compilers allow developers to write high-level programs p and transform them into low-level arithmetic circuits C, such that $[\![p]\!]_{\langle x,w\rangle} = y \iff C(\langle x, y\rangle, w)$ *is satisfiable*, where $\langle x, w\rangle$ represents the public and private inputs, respectively, and y is the program's output.

ZKP compilers generally fall into two categories: zkVM-based solutions and domain-specific languages (DSLs). zkVMs, such as SP1 [22], RISC Zero [39], and Cairo [33], typically follow a two-step process: (1) compiling the program into a standard instruction set architecture (ISA), such as RISC-V, using the established LLVM infrastructure, and (2) converting the ISA instructions into arithmetic circuits. In contrast, DSLs directly compile high-level programs (written in a DSL) into arithmetic circuits without an intermediate ISA. However, achieving both efficiency and expressiveness simultaneously remains challenging with either approach. zkVM-based solutions excel in expressiveness, which can compile nearly any Rust program into an arithmetic circuit, but they are optimized for general-purpose use and often lack tensor-specific optimizations. Besides, this process typically involves an intermediate STARK proof system and a costly translation to SNARK for on-chain deployment. Conversely, DSLs prioritize efficiency but compromise on expressiveness, and often lack support for modern programming constructs like branches and loops, as well as native tensor (or general array) operations.

Given these constraints, developers face a trade-off between expressiveness and performance when building tensor-oriented ZKP applications. Consequently, they often resort to manually crafting low-level arithmetic circuits to meet their needs [11, 20]. In this paper, we present Zinnia, an expressive and efficient zero-knowledge programming framework tailored for tensor-oriented computations. Zinnia features a DSL that allows developers to write high-level tensor operations using standard Python syntax, which are then compiled into optimized arithmetic circuits. Recall that an arithmetic circuit is fundamentally a set of polynomial equations over finite fields. The primary challenge lies in directly compiling modern programming constructs into static arithmetic circuits. To overcome this, we adopt a symbolic execution-inspired approach: we interpret program behavior in the symbolic domain, encode control-flow information (i.e., path conditions) into symbolic formulas, and generate corresponding arithmetic circuits. Unlike Python, Zinnia enforces static type checking to ensure tensor operations are well-defined at compile time. To support this, we design a custom type system and type inference mechanism to precisely define program behavior within Zinnia. To maintain flexibility akin to Python, Zinnia introduces a variable shadowing scheme, and allows users to redefine variables within the same scope. Our efforts to balance expressiveness and performance extend to the implementation. Specifically, we provide a large suite of built-in functions (e.g., Python native build-in functions, numerical operations, and tensor operators in the NumPy ecosystem) to facilitate seamless migration and development of tensor-oriented ZKP applications. For performance, we incorporate quantization for efficient real-number arithmetic and apply compilation optimizations, such as constant folding, to reduce the number of generated arithmetic constraints. In summary, we make the following contributions:

- We introduce Zinnia, an expressive and efficient tensor-oriented zero-knowledge programming framework . Zinnia features a DSL that enables developers to use modern programming constructs (e.g., dynamic control flows) within Python syntax.
- To achieve this, Zinnia adopts a symbolic execution-inspired approach to compile programs directly into optimized arithmetic circuits. It incorporates a tailored type system and variable shadowing scheme to balance expressiveness with static typing, alongside a comprehensive suite of built-in functions, quantization, and compilation optimizations.

- We evaluate Zinnia against state-of-the-art ZKP frameworks, including zkVMs and DSLs, across 25 programming tasks spanning diverse applications and a user study. Zinnia demonstrates a *statistically significant* reduction in development time, $2-3\times$ shorter code length, 19.3% smaller circuit size, and up to $245\times$ faster proving time compared to zkVMs.

2 BACKGROUND AND RELATED WORK

2.1 Zero-Knowledge Proofs (ZKPs)

ZKPs are cryptographic protocols that allow a prover to convince a verifier of the truth of a statement without revealing any information beyond its validity [17]. Among these, zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) are a family of ZKP protocols that offer succinct proofs for computational statements. These protocols enable the prover to generate a universal, non-interactive proof that is both concise and efficiently verifiable by the verifier. Currently, three prominent classes of non-interactive ZKP protocols are under active development: Groth16 [19], PLONK [9], and STARK [7]. These protocols differ in their arithmetic circuit representations, proof generation techniques, and underlying threat models. Despite these distinctions, they all operate over finite fields $\mathbb{F}_p$, based on integers and polynomials. Specifically, their arithmetic circuits are restricted to addition and multiplication operations, with each protocol imposing unique constraints on the circuit structure. Note that interactive ZKPs [37, 40], while significant, are orthogonal to our work and not reviewed here.

2.2 ZKP Library, DSL and VM

Manually constructing arithmetic circuits is tedious and error-prone. To facilitate the development of ZKPs, various programming frameworks have been introduced, including libraries, domain-specific languages (DSLs), and virtual machines (VMs). Libraries such as Halo2 [14] and LibSnark [31] provide programmatic interfaces that allow developers to construct arithmetic circuits declaratively within general-purpose languages like Rust and C++. More importantly, we introduce DSLs and VMs that offer alternative approaches to ZKP development below.

cDSLs and pDSLs for ZKP Circuit Generation. DSLs offer higher-level abstractions tailored specifically for ZKP development, enabling more intuitive circuit design. DSLs for ZKP circuit generation can be broadly categorized into constraint-based and program-based approaches [5]. Constraint-based DSLs (*cDSLs*, e.g. Circom [5]) function similarly to declarative libraries, where users explicitly define constraints that are then compiled into arithmetic circuits. Developers must also manually handle witness computation — evaluating intermediate values derived from secret inputs to ensure that circuit constraints are satisfied before generating a proof. Program-based DSLs (*pDSLs*, e.g. ZoKrates [15] and Leo [12]) provide a more expressive, high-level programming model, allowing developers to write programs that are automatically transformed into arithmetic circuits and witness computations. This approach abstracts many of the low-level details of constraint management and makes it easier to express complex logic. Despite this simplification, due to the inherent limitations of static arithmetic circuits, pDSLs often lack support for dynamic control flows and state transitions (e.g., loops, conditionals) that are common in general programming languages. Our work falls into the category of pDSLs and aims to address these limitations by providing a more expressive and efficient programming framework for tensor-oriented computations.

Zero-Knowledge Virtual Machines (zkVMs). zkVMs, such as SP1 [22], Cairo [33], and RISC0 [39], provide an alternative approach to ZKP development by allowing developers to write programs in familiar high-level languages like Rust. Instead of manually constructing arithmetic circuits, zkVMs execute programs within a software emulator of CPU instruction set. Loosely speaking, a common zkVM workflow involves first running the program to generate an execution trace, which

Table 1. Conceptual comparison with other solutions (Legend: ✓= yes, ×= no, ○= partially, ⊘= not applicable).

		Zinnia	*pDSLs*	Leo [12]	Lurk [3]	ZoKrates [15]	*ZK Programming Libraries & cDSLs*	Halo2 [14]	Circom [5]	*ZKVMs*	SP1 [22]	RISC0 [39]	Cairo [33]	*ZKML Libraries*	ezkl [20]	ZKML [11]
Control Flows	Loops	✓		✓	✓	✓		○	○		✓	✓	✓		⊘	⊘
	Loop Control	✓		×	⊘	×		○	○		✓	✓	✓		⊘	⊘
	Cond. Branch	✓		○	⊘	○		○	○		✓	✓	✓		⊘	⊘
	UDFs	✓		✓	✓	✓		○	○		✓	✓	✓		⊘	⊘
Tensor -oriented	Indicing	✓		✓	×	✓		○	○		✓	✓	✓		✓	✓
	Slicing	✓		×	×	✓		○	×		✓	✓	✓		✓	✓
	Tensors	✓		×	×	✓		○	○		✓	✓	✓		✓	✓
	Array Manip.	✓		×	×	×		×	×		×	×	×		✓	✓
	Array Comp.	✓		×	×	×		×	×		×	×	×		✓	✓
Numerical Comp.	Real Arith.	✓		×	×	×		○	○		✓	✓	✓		✓	✓
	Non-linears	✓		×	×	×		○	○		✓	✓	✓		✓	✓
Type System	Type Inference	✓		✓	✓	✓		✓	✓		✓	✓	✓		⊘	⊘
	Var. Shadow.	✓		×	×	✓		✓	×		✓	✓	✓		⊘	⊘

records the sequence of computational steps (i.e., CPU cycles). A proof is then constructed to verify that each state transition (i.e., a state consisting of the emulated CPU registers and memory) in the trace follows the expected program logic. This approach significantly enhances the usability of ZKP development by enabling developers to leverage existing programming paradigms and tools with complete control flow capabilities. However, zkVMs comes with a performance overhead due to the computational cost of CPU emulation and STARK arithmetization, as will be shown in Sec. 7.

2.3 Zero-Knowledge Machine Learning

As a closely related field, zero-knowledge machine learning (ZKML) aims to leverage ZKPs to enable secure and privacy-preserving machine learning applications. ZKML allows a model owner to prove the correctness of the inference process without disclosing the model weights. Several initiatives, such as ZKML [11, 35], ezkl [20] and ZEN [16], have explored the integration of ZKPs with machine learning. They have introduced specialized and hand-crafted circuit to accelerate machine learning primitives, e.g., the ReLU activation function, matrix multiplication, and convolutional layers. Our solution, Zinnia, is designed to complement these libraries by providing a more expressive and efficient programming framework with *automatically-optimized* tensor-oriented computations.

3 MOTIVATION AND RESEARCH OVERVIEW

3.1 Limitations of Existing Work

We present a conceptual comparison of Zinnia with other solutions in Table 1 and highlight key language features that are desirable yet currently lacking. Modern programming languages provide a rich set of control flow constructs, such as loops, conditional branches, and user-defined functions. However, both pDSL and cDSL solutions often lack full support for these constructs. For example, Leo [12] does not support loop control statements like `break` and `continue`, while Lurk relies on functional recursion to implement loops. This limitation arises from the challenge of encoding control flow logic into arithmetic circuits, which restricts the expressiveness of these languages. Efficient handling of structured data is also critical for data processing tasks. For instance, Python (with NumPy) offers convenient array access and manipulation features, allowing developers to use slicing for element access and perform advanced array computations. Such capabilities, however, are typically absent in existing ZKP solutions. Similarly, numerical computations are fundamental to data science tasks, yet most existing solutions focus on integer arithmetic and lack

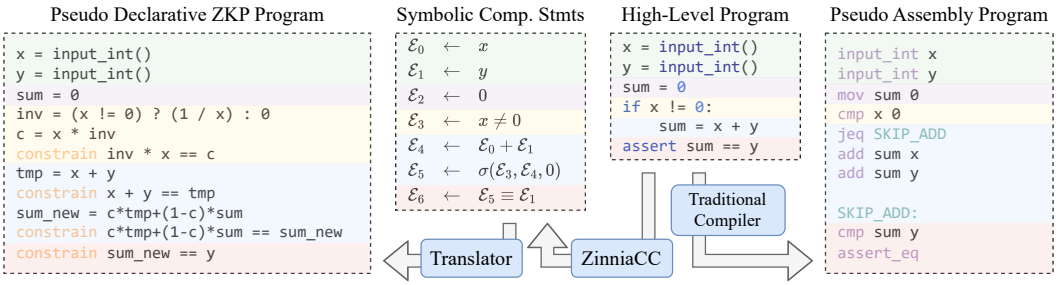

Fig. 1. **An Illustrative Example of ZINNIA's Compilation Approach.** Comparison of ZINNIA's compilation for static representation of dynamic control flows versus compilation for Turing-complete machines.

native support for real-number operations. Non-linear functions (e.g., `sqrt`, `exp`) are particularly difficult to express in these frameworks. Finally, a flexible type system is essential for developing concise and expressive code. Automatic type inference and variable shadowing, common in modern programming languages and zkVMs, are not ubiquitous in existing DSL solutions.

While zkVMs address many of these shortcomings by emulating a full CPU, they incur significant overhead due to this emulation layer and are not optimized for *data-intensive*, tensor-oriented computations. In contrast, ZKML libraries provide specialized circuits for tensor operations and numerical computations but fall short in supporting general control flow constructs. ZINNIA aims to bridge this gap by introducing a program-based DSL that prioritizes usability and expressiveness while preserving computational efficiency.

3.2 Research Overview and Technical Contributions

Study Scope. The aforementioned limitations of existing solutions motivate our research directions, which are centered around enhancing the usability, expressiveness, and efficiency of ZKP development. We particularly focus on supporting data-intensive, *tensor-oriented* operations and numerical computations (e.g. real-numbers and non-linear functions). This is important and timely, as these capabilities are essential for various machine learning and data analysis tasks, which are increasingly being applied in privacy-preserving settings. ZINNIA aims for software developers and machine learning practitioners, who are not cryptography experts, to write ZKP programs with significantly less effort and time compared to existing solutions.[2] Today, these tasks are often challenging and time-consuming due to the lack of high-level programming abstractions and the need to manually construct arithmetic circuits. Notice that we do not aim to replace zkVMs, which are suitable for general-purpose ZKP systems, but rather to complement them by providing a more efficient and expressive alternative for data-intensive computations.

Direct Compilation of Dynamic Control Flows. ZINNIA features compiling dynamic control flows to static arithmetic circuits. High-level programming languages, such as Python and Rust, provide a rich set of control flow constructs. As aforementioned, zkVMs support these control-flow constructs at the cost of an additional emulation layer. In ZINNIA, we advocate for a direct compilation approach of these constructs to static arithmetic circuits while alleviating the performance overhead associated with zkVMs. We anticipate that this approach will not only enhance the usability and expressiveness of ZKP development but also maintains the efficiency of SNARK-based systems by avoiding emulation overhead in zkVMs.

Despite the promising prospects of direct compilation, the transformation of dynamic control flows to static arithmetic circuits is non-trivial. To understand it, we present an illustrative example

[2]In other words, we do not require users of ZINNIA to understand the underlying cryptographic protocols.

in Figure 1. In this example, when being compiled directly to assembly program (rightmost side of Figure 1), the control flow can be easily represented by multiple jump instructions. However, when compiled to arithmetic circuits (leftmost side of Figure 1), representing such control flow (which is often dynamic and runtime-dependent) poses a fundamental challenge. This is because arithmetic circuits, which encode computations as a fixed directed acyclic graph, does not provide jump instructions where we cannot inherently accommodate the dynamic nature of control flows.

One key insight of our approach draws upon *symbolic execution* [4, 21], a program analysis technique that systematically explores all feasible execution paths using symbolic rather than concrete values. By encapsulating the dynamic semantics of high-level programs within a unified static representation, Zinnia effectively consolidates execution paths into a single static arithmetic circuit. For instance, as depicted in Figure 1, our methodology leverages symbolic execution to capture the path conditions of conditional branches and employs a symbolic selection expression $\sigma(c, x, y) = cx + (1 - c)y$ to compactly represent conditional assignments (i.e., `if c then x else y`) in the generated circuits. This symbolic representation ensures that the generated circuit accurately reflects the original program's behavior. Those representations can be easily compiled into arithmetic circuits, thereby bridging the gap. Shortly in Section 5, we will provide a principled formalism for this approach that recursively compiles the control flow constructs into arithmetic circuits; and in Sec. 7, our empirical evaluation validates its superiority over zkVMs with at least 25.4% and 5.0% advantages in proving and verifying times, respectively.

Usability and Expressiveness. As shown in Table 1, Zinnia provides a comprehensive set of control flow constructs. This enables developers to articulate complex computational logic in a familiar and intuitive manner. Moreover, by incorporating advanced tensor-oriented and numerical computation functionalities, Zinnia features the ever first support for data-intensive, tensor-oriented computations in ZKP development. The type system of Zinnia further enhances the development workflow by promoting concise and expressive code. We view these features as essential for offering high usability and expressiveness in ZKP development. As will be shown in Sec. 7, Zinnia supports a wide range of real applications that are frequently encountered in practice yet challenging to implement using existing solutions, including machine learning, data analysis, and cryptographic protocols. Human study in Sec. 7 further validates the usability and expressiveness of Zinnia by demonstrating that developers can write complex ZKP programs with significantly less effort and time compared to existing solutions.

Efficiency. Zinnia offers high computational efficiency, generating compact and optimized arithmetic circuits. This can be attributed to the *direct compilation paradigm* and the integration of a suite of *compiler Intermediate Representation (IR)-level optimizations*. Apparently, the direct compilation paradigm can principally eliminate the performance overhead associated with zkVMs, as it directly compiles dynamic control flows to static arithmetic circuits. We perform further optimizations (e.g., constant folding and symbolic state simplification) over the intermediate symbolic representation to reduce redundant operations and streamline the overall circuit structure. Moreover, a direct compilation methodology alone does not yield the optimal circuits. Similar to standard program compilation pipeline, it necessitates optimization passes to reduce redundant operations and streamline the overall circuit structure. To this end, we follow the pipeline of traditional compilers and introduce an IR to represent the computation statements in the static single assignment (SSA) form. On top of this IR, we implement data flow analysis and a suite of standard optimization passes, such as constant propagation, dead-code elimination, and common subexpression elimination detailed in Sec. 5. As we will demonstrate in Sec. 7, Zinnia outperforms existing solutions in both constraint count and proving time, achieving a 19.3% reduction in constraint count and a 4.9% reduction in proving time compared to the Halo2 baseline. Our ablation study further highlights the impact of these optimizations, showing a 46.0% increase in constraint count when they are

removed. Additionally, ZINNIA's direct compilation approach proves to be faster than the fastest zkVMs over 25.4× in proving time and over 5.0× faster in verification time.

4 THE ZINNIA PROGRAMMING FRAMEWORK

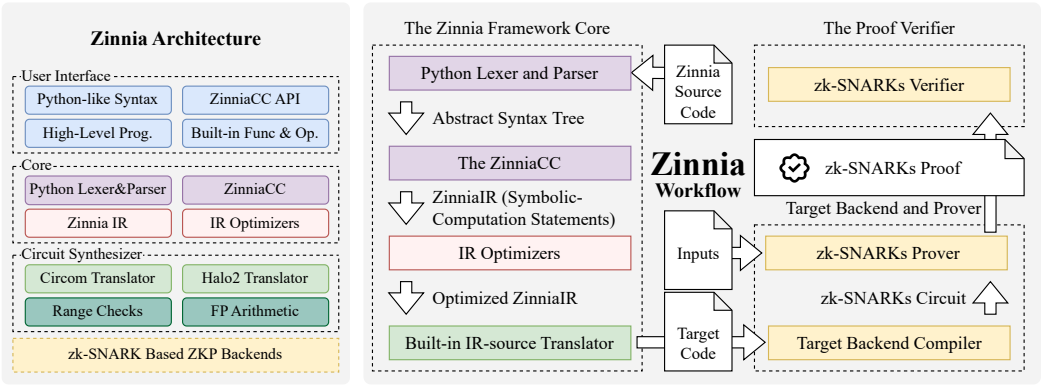

Fig. 2. **The Architecture and Workflow Diagram of ZINNIA.**

ZINNIA Architecture. The architectural design of ZINNIA, depicted in Figure 2, is structured into three primary layers: the User Interface, the ZINNIA Core, and the Circuit Synthesizer. At the topmost layer, the ZINNIA's User Interface provides a Python-compatible syntax for developing ZKP programs. It enables users to write high-level ZINNIA programs with dynamic control-flow constructs (e.g., branching and looping) and user-defined functions. As detailed in Sec. 6, ZINNIA implements a set of commonly-used built-in functions and tensor operators from the Python and NumPy ecosystems, enhancing usability for data science and machine learning tasks.

With the user-written program, the Core layer parses the program, constructs the abstract syntax tree (AST), and symbolically executes it to generate an Intermediate Representation (ZINNIAIR). The IR represents the program's computations symbolically in SSA form, on which optimization techniques are applied to minimize the number of IR statements and enhance performance.

Finally, the Circuit Synthesizer layer generates low-level source code for target backends from ZINNIAIR, including Halo2 [14] (with experimental support for Circom [5]). This layer also implements real number arithmetic and supports range checks such that certain intrinsic functions in IR can be represented in an efficient and compact form. The generated code is then passed to the respective backends to produce arithmetic circuits for the zk-SNARKssystem.

ZINNIA Workflow. As depicted in the right side of Figure 2, ZINNIA can be smoothly integrated into existing development pipelines for privacy purposes. Users, such as data scientists and developers in a privacy-sensitive domain, can write ZINNIA programs for their privacy-preserving machine learning and data analytics tasks. The ZINNIACC compiler translates these programs into ZINNIAIR, which is then optimized by IR Optimizers. The Circuit Synthesizer generates backend-specific code, which is compiled into arithmetic circuits by the target backend (e.g. Halo2). Then, with the compiled circuits, users can provide witness inputs to generate proofs and verification keys with the help of a ZKP prover. Lastly, the generated proofs can be verified a ZKP verifier. Notice that both the prover and verifier systems are not part of the ZINNIA framework. Currently, ZINNIA supports the widely adopted Halo2 prover and verifier systems [14]. However, ZINNIA is designed to be backend-agnostic, with the flexibility to support additional ZKP systems in the future (e.g., experimental support is provided for Circom [5]).

Program	P	::=	$\epsilon \mid s; P$
Statement	s	::=	$s_1; s_2 \mid e \mid \texttt{if}\ e : s_1\ \texttt{else} : s_2 \mid \texttt{if}\ e : s$
		|	$\texttt{for}\ id\ \texttt{in}\ e : s \mid \texttt{while}\ e : s \mid \texttt{continue} \mid \texttt{break} \mid \texttt{pass}$
		|	$\texttt{return}\ e \mid \texttt{assert}\ e \mid id \leftarrow e \mid e_1[e_2] \leftarrow e_3 \mid s^{\text{input}} \mid s^{\text{func}}$
Func. Declare	s^{func}	::=	$id^{\text{func}}(id_1 : t_1, id_2 : t_2, \ldots, id_N : t_N) \rightarrow t_r : s$
Input Stmt.	s^{input}	::=	$\texttt{input\_public}(id, t) \mid \texttt{input\_private}(id, t) \mid \texttt{input\_hashed}(id, t)$
Literal	c	::=	INTEGER | FLOAT | True | False | None
Type	t	::=	$\texttt{Integer} \mid \texttt{Boolean} \mid \texttt{Float} \mid \texttt{None} \mid \texttt{Tensor}[t, (e_1, e_2, \ldots, e_N)]$
		|	$\texttt{List}[t_1, t_2, \ldots, t_N] \mid \texttt{Tuple}[t_1, t_2, \ldots, t_N]$
Expression	e	::=	$id \mid c \mid t \mid e_1 \otimes e_2 \mid \odot e \mid e_1[e_2] \mid [e_1, e_2, \ldots, e_N]$
		|	$(e_1, e_2, \ldots, e_N) \mid id(e_1, e_2, \ldots, e_N) \mid op(e_1, e_2, \ldots, e_N)$
Binary Op.	$\otimes$	::=	$+ \mid - \mid * \mid / \mid \% \mid ** \mid @ \mid // \mid \texttt{and} \mid \texttt{or}$
Unary Op.	$\odot$	::=	$+ \mid - \mid \texttt{not}$
Built-in Op.	op	::=	list | tuple | range | iter | get_item | new | hash | ...
Identifier	id	::=	IDENTIFIER

Fig. 3. **The Syntax of Zinnia.**

Zinnia Language. The Zinnia DSL is specifically designed to simplify the development of ZKP programs within the zk-SNARKs context. By offering dynamic control-flow constructs and built-in functions, Zinnia seeks to provide both efficiency and a programming environment that is familiar to developers. We present the syntax of our DSL in Figure 3, which is generally compatible with Python and NumPy conventions, making the language both familiar and intuitive for programmers, data engineers, and machine learning practitioners. In other words, Zinnia programs are also valid Python programs, but not vice versa. To make the resulting circuit manageable and efficient for ZKP systems, we make a few modifications and restrictions as follows.

First, Zinnia requires explicit type annotations for the parameters and return values of all functions, which allow us to decide suitable circuit representations at compile time. Second, Zinnia supports a limited set of base types in comparison to Python, including `Integer`, `Float`, `Boolean`, and `None`. However, characters, strings, and bytes are not supported as the mainstream applications of ZKP systems do not require them and representing string/byte manipulation in circuits is very costly. Likewise, Zinnia do not offer support for advanced data structures such as dictionaries and sets. When these base types or data structures are needed, we recommend users to preprocess the data before feeding it into Zinnia. Third, since Zinnia has to be compiled into static arithmetic circuits, it does not support dynamic memory allocation. In that sense, users must declare the size of tensors and arrays at compile time. Finally, a ZKP program typically requires a set of public inputs, private inputs, and hashed inputs. In Zinnia, we offer `Public/Private/Hashed` for handy annotations, which will be used by the compiler to determine the visibility of the inputs.

In summary, the DSL is tailored to provide a familiar and expressive programming environment for developing ZKP programs. In the next section, we will show how to compile programs written in this DSL into arithmetic circuits using the ZinniaCC compiler.

5 THE ZINNIA CIRCUIT COMPILER (ZINNIACC)

As noted in Sec. 3, ZinniaCC employs a symbolic execution-inspired approach to translating dynamic program logic into the static arithmetic circuits required for SNARKs. Below, we first introduce the symbolic state managed by ZinniaCC in Sec. 5.1, which maintains the program's state symbolically. Then, in Sec. 5.2, we introduce the operational semantics of our DSL and show how ZinniaCC constructs static, fixed-flow computation statements based on the symbolic state, which is represented by ZinniaIR and would be translated into declarative ZKP programs including

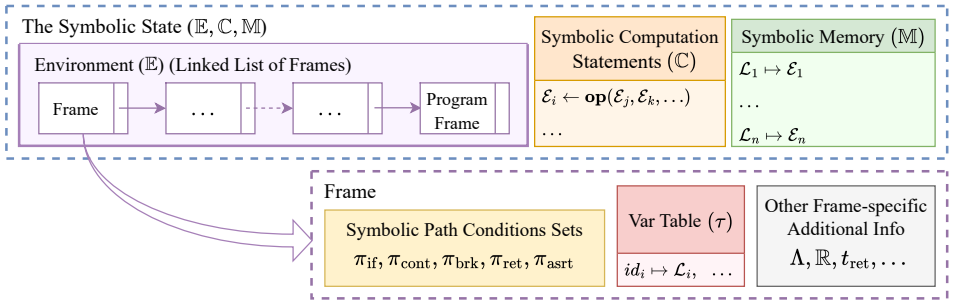

Fig. 4. **Components of ZinniaCC's Symbolic State.** This figure uses $\mathcal{E}$ to denote symbolic expressions, $\mathcal{L}$ for memory locations, **op** for computational operators, and *id* for identifiers. Further notations will be explained as the components are detailed.

libraries and cDSLs. Finally, we discuss the type system of ZinniaCC with built-in types, static typing mechanism, and type inference capabilities with variable shadowing feature in Sec. 5.3.

5.1 The Symbolic State

In a typical program execution, the "state" represents the values of all variables, the program counter, and the program memory. The state evolves as the program executes, and reflects the program's control flow and data manipulation [32]. In a similar vein, Zinnia maintains a state at compile time while it is symbolic in nature. As shown in Figure 4, the symbolic state includes *Environment* ($\mathbb{E}$), *Symbolic Memory* ($\mathbb{M}$), and *Computation Statements Set* ($\mathbb{C}$). In a nutshell, the Environment models the program's scope and control flow, which allows ZinniaCC to track the program's state without concrete inputs. The Symbolic Memory maps memory addresses to symbolic expressions, and the Computation Statements Set packs all symbolic expressions as the compilation result. In the following, we elaborate on each component of the symbolic state. The symbolic state is illustrated in Figure 4, and we will detail each component below.

Environment $\mathbb{E}$. The environment $\mathbb{E}$ models the program's scope and is structured as a linked list of frames [1]. It resolves variable names and tracks the program's control flow. Each frame represents a distinct scope of the program. With a sequence of linked frames, ZinniaCC can navigate through the program's scopes and ensure accurate variable resolution.

Frame. Frames are data structures that represent program scopes. Each frame represents a distinct scope and contains a variable table (τ), which maps variable names to their corresponding memory locations. Additionally, each frame stores path conditions that must be satisfied for the program to reach the point represented by that frame. The parent frame of the current frame is denoted as $\hat{\mathbb{E}}$. Zinnia defines several frame types, including `ProgramFrame`, `IfFrame`, `LoopFrame`, and `FuncFrame`. `ProgramFrame` is the root-level frame, which includes a table (Λ) of user-defined functions and a public inputs set $\mathbb{P}$. `IfFrame` and `LoopFrame` store the condition expressions for the corresponding control flow constructs. `FuncFrame` stores the function's return type (t_{ret}) and a list of return values with corresponding conditions ($\mathbb{R}$). When resolving a variable, ZinniaCC searches the current frame first, then recursively searches parent frames until the variable is found or the top-level frame is reached.

Path Condition. Path conditions [4], stored within frames, are symbolic expressions that must evaluate to true for the program to reach a specific point. They capture the control flow of the program. Zinnia supports several types of path conditions: if conditions (π_{if}), break conditions (π_{brk}), continue conditions (π_{cont}), and return conditions (π_{ret}). The overall condition, $\pi^{\mathbb{E}} = (\bigwedge_{\mathcal{E} \in \pi^{\mathbb{E}}_{\text{if}}} \mathcal{E}) \wedge (\bigwedge_{\mathcal{E} \in \pi^{\mathbb{E}}_{\text{brk}} \cup \pi^{\mathbb{E}}_{\text{cont}} \cup \pi^{\mathbb{E}}_{\text{ret}}} \neg\mathcal{E})$ is formed by combining the if conditions with the negated break, continue,

and return conditions. While π_{if} represents the condition for the if statement to be executed, π_{brk}, π_{cont}, and π_{ret} represent the conditions associated with loops and functions to be skipped. Thus, path conditions in π_{brk}, π_{cont}, and π_{ret} are negated to express an expression that must be true for the program to reach the current point. Additionally, an assertion condition (π_{asrt}) enables or disables assertions, requiring it to be true for assertions to be active.

Variable Table τ. Each frame contains a variable table, a mapping from variable names to their memory locations. The set of all variables within environment $\mathbb{E}$ is denoted as $\tau^{\mathbb{E}}$, and the memory location of variable id is denoted as $\tau^{\mathbb{E}}_{id}$.

Symbolic Computation Statements $\mathbb{C}$. The computation statements set $\mathbb{C}$ represents the compilation result, a list of symbolic computations that are evaluated using concrete values during zk-SNARKs circuit proving and verified as arithmetic constraints during zk-SNARKs circuit verifying. These computation statements serve as intermediate representations, generated throughout the compilation process, and the final set forms the output of the compilation. We formulate accumulated statements as $\mathbb{C}$ as an integral part of the symbolic state to reflect the accumulated compilation result.

Symbolic Memory $\mathbb{M}$ and Memory Model. The symbolic memory $\mathbb{M}$ is a mapping from memory addresses to symbolic expressions. It is crucial for handling built-in in-place functions (e.g. the `set_item` function for supporting subscript assign $id[e] \leftarrow e$), which modify the symbolic expressions of input variables. A symbolic evaluator without memory who evaluates expressions and statements recursively in a stateless manner would not be capable of such modifications.

We employ a sound memory model to manage the symbolic memory access and update [13]. To do so, pointer aliasing, specifically within array indexing operations, is addressed through multiplexing (conditional selection). Consider pointers `pa` and `pb` that may alias. Upon updating the value referenced by `pa`, a multiplexing operation updates all potential locations pointed to by `pa` with a conditional expression: the new value if the location's index equals `pa`, and the existing value otherwise. Subsequently, accessing the value referenced by `pb` employs a similar multiplexing operation, selecting the appropriate value based on the equality between the location's index and `pb`. This methodology ensures sound and accurate propagation of updates within the symbolic memory, effectively managing pointer aliasing. It however do not incur a significant overhead per our observation, as the number of multiplexing operations is bounded by the number of memory locations, which is typically small in ZKP programs and limited to array indexing operations.

5.2 Computation Statement Generation

We now present how ZinniaCC symbolically captures program behavior and subsequently generates computation statements that reflect this behavior within the circuit. Specifically, we will delineate ZinniaCC's symbolic evaluations of expressions, handling of symbolic inputs, and processing of various control flow constructs and other non-control flow statements through operational semantics. Finally, we discuss user-defined functions as a key feature in high-level programming languages and how ZinniaCC handles them.

Notations. To formalize our program semantics, we use $\mathbb{E}, \mathbb{C}, \mathbb{M}$ to denote the symbolic state, where $\mathbb{E}$ is the environment, $\mathbb{C}$ is the computation statements set, and $\mathbb{M}$ is the symbolic memory. Then, we use inference rules denoted as $\mathbb{E}, \mathbb{C}, \mathbb{M} \vdash \{e\} \Downarrow \mathbb{E}', \mathbb{C}', \mathbb{M}', \langle \mathcal{L}, \mathcal{E}, \theta \rangle$. This notation denotes how ZinniaCC processes a program construct $\{e\}$, transitioning from symbolic states $\mathbb{E}$, $\mathbb{C}$, $\mathbb{M}$ to $\mathbb{E}'$, $\mathbb{C}'$, $\mathbb{M}'$. Each expression evaluates to $\langle \mathcal{L}, \mathcal{E}, \theta \rangle$, where $\mathcal{L}$ is the memory location, $\mathcal{E}$ is the symbolic expression, and θ is the type mutable indicator. Type consistency, which is used to support variable shadowing, enforced by $\theta \in \{\dagger, \ddagger\}$ and the type mutable indicator function Θ, is detailed in Subsection 5.3. This triplet is omitted for program constructs without an evaluation result.

5.2.1 Expression Evaluation. As depicted in Figure 5, ZinniaCC evaluates expressions, including variable references, literals, and operators, into symbolic expressions. In Var-Load-Expr, we evaluate a variable reference by first looking up the variable's memory location in the environment and symbolic memory, then retrieving the corresponding symbolic expression, and finally computing the type mutable indicator for this identifier. In Literal-Expr, we evaluate a literal by directly assigning the literal value to a new memory location and updating the memory. In Binary-Expr, we evaluate a binary expression by recursively evaluating the left and right operands, then applying the corresponding operator to the operands' symbolic expressions. A new memory location is obtained, and the result is stored in the memory. In Unary-Expr, the evaluation process is similar to Binary-Expr, but for unary operators.

Var-Load-Expr

$$\frac{id \in \tau^{\mathbb{E}} \quad \mathcal{L} = \tau^{\mathbb{E}}_{id} \quad \theta = \Theta^{\mathbb{E}}_{id} \quad \mathbb{M}(\mathcal{L}) = \mathcal{E}}{\mathbb{E}, \mathbb{C}, \mathbb{M} \vdash \{id\} \Downarrow \mathbb{E}, \mathbb{C}, \mathbb{M}, \langle \mathcal{L}, \mathcal{E}, \theta \rangle}$$

Literal-Expr

$$\frac{\mathcal{L} = |\mathbb{M}| \quad \mathbb{M}' = \mathbb{M}[\mathcal{L} \mapsto c]}{\mathbb{E}, \mathbb{C}, \mathbb{M} \vdash \{c\} \Downarrow \mathbb{E}, \mathbb{C}, \mathbb{M}', \langle \mathcal{L}, c, \dagger \rangle}$$

Binary-Expr

$$\frac{\begin{array}{c}\mathbb{E}_0, \mathbb{C}_0, \mathbb{M}_0 \vdash \{e_1\} \Downarrow \mathbb{E}_1, \mathbb{C}_1, \mathbb{M}_1, \langle \mathcal{L}_1, \mathcal{E}_1, \theta_1 \rangle \quad \mathbb{E}_1, \mathbb{C}_1, \mathbb{M}_1 \vdash \{e_2\} \Downarrow \mathbb{E}_2, \mathbb{C}_2, \mathbb{M}_2, \langle \mathcal{L}_2, \mathcal{E}_2, \theta_2 \rangle \\ \mathcal{E}' = \mathcal{E}_1 \otimes \mathcal{E}_2, \quad \mathcal{L}_3 = |\mathbb{M}_2| \quad \mathbb{M}_3 = \mathbb{M}_2[\mathcal{L}_3 \mapsto \mathcal{E}']\end{array}}{\mathbb{E}_0, \mathbb{C}_0, \mathbb{M}_0 \vdash \{e_1 \otimes e_2\} \Downarrow \mathbb{E}_2, \mathbb{C}_1 \cup \mathbb{C}_2 \cup \{\mathcal{E}' \leftarrow \mathcal{E}_1 \otimes \mathcal{E}_2\}, \mathbb{M}_3, \langle \mathcal{L}_3, \mathcal{E}', \dagger \rangle}$$

Unary-Expr

$$\frac{\mathbb{E}, \mathbb{C}, \mathbb{M} \vdash \{e_1\} \Downarrow \mathbb{E}_1, \mathbb{C}_1, \mathbb{M}_1, \langle \mathcal{L}_1, \mathcal{E}_1, \theta_1 \rangle \quad \mathcal{E}_2 = \odot\, \mathcal{E}_1 \quad \mathcal{L}_2 = |\mathbb{M}_1| \quad \mathbb{M}_2 = \mathbb{M}_1[\mathcal{L}_2 \mapsto \mathcal{E}_2]}{\mathbb{E}, \mathbb{C}, \mathbb{M} \vdash \{\odot\, e\} \Downarrow \mathbb{E}_1, \mathbb{C}_1 \cup \{\mathcal{E}_1 \leftarrow \odot\, \mathcal{E}_2\}, \mathbb{M}_2, \langle \mathcal{L}_2, \mathcal{E}_2, \dagger \rangle}$$

Fig. 5. **Operational Semantics for Expression Evaluations.**

5.2.2 Symbolic Input. ZinniaCC handles symbolic inputs by assigning symbolic representations for each input variable. Based on the nature of arithmetic circuits, we define three types of input statements as shown in Figure 6. Input-Stmt (Public) is used for inputting public variables, which are known to both the prover and verifier. We store the corresponding symbolic expressions in the public inputs set $\mathbb{P}$ within the environment. Input-Stmt (Private) is used for inputting private variables, which are known only to the prover. Input-Stmt (Hashed) is used for inputting hashed variables, which are hashed before being input to the circuit. We insert additional hash computations to ensure the correctness of the hashed inputs. The hash value is treated as a public input. In these input statements, we use α^t_{id} to denote the symbolic input for variable id with type t.

Input-Stmt (Public)

$$\frac{\mathcal{E} = \alpha^t_{id} \quad \mathcal{L} = |\mathbb{M}| \quad \mathbb{M}' = \mathbb{M}[\mathcal{L} \mapsto \mathcal{E}]}{\begin{array}{c}\mathbb{E}, \mathbb{C}, \mathbb{M} \vdash \{\texttt{input\_public}(id, t)\} \Downarrow \\ \mathbb{E}[id \mapsto \mathcal{L}], \mathbb{C}, \mathbb{M}', \langle \mathcal{L}, \mathcal{E}, \dagger \rangle\end{array}}$$

Input-Stmt (Private)

$$\frac{\mathcal{E} = \alpha^t_{id} \quad \mathcal{L} = |\mathbb{M}| \quad \mathbb{M}' = \mathbb{M}[\mathcal{L} \mapsto \mathcal{E}]}{\begin{array}{c}\mathbb{E}, \mathbb{C}, \mathbb{M} \vdash \{\texttt{input\_private}(id, t)\} \Downarrow \\ \mathbb{E}[id \mapsto \mathcal{L}, \mathbb{P} \leftarrow \mathbb{P} \cup \{\mathcal{E}\}], \mathbb{C}, \mathbb{M}', \langle \mathcal{L}, \mathcal{E}, \dagger \rangle\end{array}}$$

Input-Stmt (Hashed)

$$\frac{\mathcal{E} = \alpha^t_{\text{id}} \quad \mathcal{L} = |\mathbb{M}| \quad \mathcal{E}_{\text{hash}} = \alpha^{\text{Hashed}[t]}_{id} \quad \mathbb{E}[id \mapsto \mathcal{L}], \mathbb{C}, \mathbb{M}[\mathcal{L} \mapsto \mathcal{E}] \vdash \{\text{hash}(id)\} \Downarrow \mathbb{E}', \mathbb{C}', \mathbb{M}' \langle \mathcal{L}', \mathcal{E}', \theta \rangle}{\mathbb{E}, \mathbb{C}, \mathbb{M} \vdash \{\texttt{input\_hashed}(id, t)\} \Downarrow \mathbb{E}'[\mathbb{P} \leftarrow \mathbb{P} \cup \{\mathcal{E}_{\text{hash}}\}], \mathbb{C}' \cup \{\mathcal{E}' \equiv \mathcal{E}_{\text{hash}}\}, \mathbb{M}', \langle \mathcal{L}, \mathcal{E}, \dagger \rangle}$$

Fig. 6. **Operational Semantics for Symbolic Inputs.**

$$\textsc{If-Else-Stmt}\quad \frac{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{e_1\}\Downarrow\mathbb{E}_1,\mathbb{C}_1,\mathbb{M}_1,\langle\mathcal{L},\mathcal{E},\theta\rangle \quad \text{IfFrame}(\mathbb{E}_1,\mathcal{E}),\mathbb{C}_1,\mathbb{M}_1\vdash\{s_1\}\Downarrow\mathbb{E}_2,\mathbb{C}_2,\mathbb{M}_2 \quad \text{IfFrame}(\hat{\mathbb{E}_2},\neg\mathcal{E}),\mathbb{C}_2,\mathbb{M}_2\vdash\{s_2\}\Downarrow\mathbb{E}',\mathbb{C}',\mathbb{M}'}{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{\texttt{if } e_1:s_1 \texttt{ else}:s_2\}\Downarrow\hat{\mathbb{E}'},\mathbb{C}',\mathbb{M}'}$$

$$\textsc{If-Stmt}\quad \frac{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{e_1\}\Downarrow\mathbb{E}_1,\mathbb{C}_1,\mathbb{M}_1,\langle\mathcal{L},\mathcal{E},\theta\rangle \quad \text{IfFrame}(\mathbb{E}_1,\mathcal{E}),\mathbb{C}_1,\mathbb{M}_1\vdash\{s_1\}\Downarrow\mathbb{E}',\mathbb{C}',\mathbb{M}'}{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{\texttt{if } e_1:s_1\}\Downarrow\hat{\mathbb{E}'},\mathbb{C}',\mathbb{M}'}$$

$$\textsc{For-In-Stmt}\quad \frac{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{\texttt{iter}(e)\}\Downarrow\mathbb{E}^*,\mathbb{C}_0,\mathbb{M}_0,\{\langle\mathcal{L}_1,\mathcal{E}_1,\theta_1\rangle,\ldots,\langle\mathcal{L}_i,\mathcal{E}_i,\theta_i\rangle,\ldots,\langle\mathcal{L}_N,\mathcal{E}_N,\theta_N\rangle\} \quad \mathbb{E}_0=\text{LoopFrame}(\mathbb{E}^*) \quad \mathbb{E}_{i-1}[id\mapsto\mathcal{L}_i,\pi_{\text{cont}}\leftarrow\emptyset],\mathbb{C}_{i-1},\mathbb{M}_{i-1}\vdash\{s\}\Downarrow\mathbb{E}_i,\mathbb{C}_i,\mathbb{M}_i, \text{where } i\in\{1,2,\ldots,N\}}{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{\texttt{for } id \texttt{ in } e:s\}\Downarrow\hat{\mathbb{E}_N},\mathbb{C}_N,\mathbb{M}_N}$$

$$\textsc{While-Stmt}\quad \frac{\mathbb{E}_0=\text{LoopFrame}(\mathbb{E}) \quad \mathbb{E}_{i-1},\mathbb{C}_{i-1},\mathbb{M}_{i-1}\vdash\{e\}\Downarrow\mathbb{E}'_{i-1},\mathbb{C}'_{i-1},\mathbb{M}'_{i-1},\langle\mathcal{L}_i,\mathcal{E}_i,\theta_i\rangle \quad \mathbb{E}'_{i-1}[\pi_{\text{cont}}\leftarrow\emptyset,\pi_{\text{brk}}\leftarrow\pi_{\text{brk}}\cup\pi\cup\{\mathcal{E}_i\}],\mathbb{C}'_{i-1},\mathbb{M}'_{i-1}\vdash\{s\}\Downarrow\mathbb{E}_i,\mathbb{C}_i,\mathbb{M}_i, \text{where } i\in\{1,2,\ldots,\mu\}}{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{\texttt{while } e:s\}\Downarrow\hat{\mathbb{E}_\mu},\mathbb{C}_\mu,\mathbb{M}_\mu}$$

$$\textsc{Continue-Stmt}\quad \frac{\text{InLoop}(\mathbb{E})}{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{\texttt{continue}\}\Downarrow\mathbb{E}[\pi_{\text{cont}}\leftarrow\pi_{\text{cont}}\cup\pi],\mathbb{C},\mathbb{M}}$$

$$\textsc{Break-Stmt}\quad \frac{\text{InLoop}(\mathbb{E})}{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{\texttt{break}\}\Downarrow\mathbb{E}[\pi_{\text{brk}}\leftarrow\pi_{\text{brk}}\cup\pi],\mathbb{C},\mathbb{M}}$$

$$\textsc{Pass-Stmt}\quad \frac{}{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{\texttt{pass}\}\Downarrow\mathbb{E},\mathbb{C},\mathbb{M}}$$

$$\textsc{Return-Stmt}\quad \frac{\text{InFunc}(\mathbb{E}) \quad \mathbb{E},\mathbb{C},\mathbb{M}\vdash\{e\}\Downarrow\mathbb{E}',\mathbb{C}',\mathbb{M}',\langle\mathcal{L},\mathcal{E},\theta\rangle \quad \mathcal{E}^*=\pi^{\mathbb{E}} \quad \Gamma(\mathcal{E})=t^{\mathbb{E}}_{\text{ret}} \quad \mathbb{E}''=\text{UpdateBreakConditionsRecursively}(\mathbb{E}'[\pi_{\text{ret}}\leftarrow\pi_{\text{ret}}\cup\pi,\mathbb{R}\leftarrow\mathbb{R}\cup(\mathcal{E}^*,\mathcal{E})],\mathcal{E}^*)}{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{\texttt{return } e\}\Downarrow\mathbb{E}'',\mathbb{C}',\mathbb{M}'}$$

Fig. 7. **Operational Semantics for Control Flow Constructs.**

5.2.3 Control Flow Statements. The methods to capture every execution result expressed by control flows are a crucial part of ZinniaCC. We present how ZinniaCC handles conditional flows using the semantic rules in Figure 7.

Conditional Branch. If-Else-Stmt evaluates the condition expression e_1 to obtain the symbolic expression for the branching condition. It then creates a new frame using the `IfFrame` function, storing the condition expression in the frame's path condition. ZinniaCC then evaluates the statements in the true branch. Subsequently, another new `IfFrame` is created for the false branch, and the statements in the false branch are evaluated. The path condition for the false branch is the negation of the condition expression. If-Stmt is similar to If-Else-Stmt without the false branch. Specifically, ZinniaCC will skip infeasible branches for the seek of speeding up our compilation if the symbolic condition is evaluated to be always unsatisfiable.

Loop. Loops are another essential control flow constructs which will be unrolled into fixed-flow computations in Zinnia. In For-In-Stmt, We use the operator `iter`(e) to evaluate the iterable object and obtain the symbolic expressions for each element. By nature, the number of iterations is known for a For-In-Stmt loop since we are working with a statically typed language. We create a new `LoopFrame` to store the loop's path condition, update the mapping for the identifier, and evaluate the loop body for each iteration. For While-Stmt, we also create a new `LoopFrame` to store the loop's path condition and evaluate the loop body until the condition is false or the loop is broken. For each iteration, we update the break condition with the evaluation result of the loop condition e. However, as arithmetic circuits are static, we must ensure that the loop terminates within a fixed number of iterations. As the halting problem is undecidable [34], we track the number

$$\textsc{Assert-Stmt}\quad \frac{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{e\}\Downarrow\mathbb{E}',\mathbb{C}',\mathbb{M}',\langle\mathcal{L},\mathcal{E},\theta\rangle \quad \mathcal{E}^{*1}=\pi^{\mathbb{E}} \quad \mathcal{E}^{*2}=\pi^{\mathbb{E}}_{\text{asrt}}}{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{\texttt{assert}\ e\}\Downarrow\mathbb{E}',\mathbb{C}'\cup\{\neg(\mathcal{E}^{*1}\wedge\mathcal{E}^{*2})\vee\mathcal{E}\},\mathbb{M}'}$$

$$\textsc{Assign (New)}\quad \frac{id\notin\tau^{\mathbb{E}} \quad \mathbb{E},\mathbb{C},\mathbb{M}\vdash\{e\}\Downarrow\mathbb{E}',\mathbb{C}',\mathbb{M}',\langle\mathcal{L},\mathcal{E},\theta\rangle}{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{id\leftarrow e\}\Downarrow\mathbb{E}'[id\mapsto\mathcal{L}],\mathbb{C}',\mathbb{M}'}$$

$$\textsc{Assign (Existing, Mut)}\quad \frac{id\in\tau^{\mathbb{E}} \quad \dagger=\Theta^{\mathbb{E}}_{id} \quad \mathbb{E},\mathbb{C},\mathbb{M}\vdash\{e\}\Downarrow\mathbb{E}',\mathbb{C}',\mathbb{M}',\langle\mathcal{L},\mathcal{E},\theta\rangle}{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{id\leftarrow e\}\Downarrow\mathbb{E}'[id\mapsto\mathcal{L}],\mathbb{C}',\mathbb{M}'}$$

$$\textsc{Assign (Existing, Immut)}\quad \frac{\begin{array}{c}id\in\tau^{\mathbb{E}} \quad \ddagger=\Theta^{\mathbb{E}}_{id} \quad \mathcal{L}=\tau^{\mathbb{E}}_{id} \quad \mathcal{E}=\mathbb{M}(\mathcal{L}) \quad \mathbb{E},\mathbb{C},\mathbb{M}\vdash\{e\}\Downarrow\mathbb{E}',\mathbb{C}',\mathbb{M}',\langle\mathcal{L}',\mathcal{E}',\theta\rangle \quad \mathcal{E}^{*}=\pi^{\mathbb{E}}\\ \Gamma(\mathcal{E})=\Gamma(\mathcal{E}') \quad \mathcal{E}''=\sigma(\mathcal{E}^{*},\mathcal{E}',\mathcal{E}) \quad \mathcal{L}''=|\mathbb{M}'| \quad \mathbb{M}''=\mathbb{M}'[\mathcal{L}''\leftarrow\mathcal{E}'']\end{array}}{\mathbb{E},\mathbb{C},\mathbb{M}\vdash\{id\leftarrow e\}\Downarrow\mathbb{E}'[id\mapsto\mathcal{L}''],\mathbb{C}'\cup\{\mathcal{E}''\leftarrow\sigma(\mathcal{E}^{*},\mathcal{E}',\mathcal{E})\},\mathbb{M}''}$$

Fig. 8. **Operational Semantics for Non-Control Flow Statements.**

of iterations and break the loop with a warning notifying the user if the number of iterations exceeds a predefined limit μ, which is a user configurable parameter. For each of these two loop constructs, we clear the continue condition set π_{cont} each iteration.

Loop Control Statement. Loop Control Statements, such as `break` and `continue`, are used to alter the control flow of a program. Continue-Stmt obtains the path condition for the current program execution point, $\pi^{\mathbb{E}}$, and adds the condition to the continue condition set π_{cont}. Break-Stmt is similar to Continue-Stmt, but adds the condition to the break condition set π_{brk}. Pass-Stmt does not alter the control flow and is used as a placeholder for empty blocks.

Return. Zinnia's design allows for the definition of user-defined functions, which can be called within the program. The return statement is used to return a value from a function or to terminate the function's execution. In Return-Stmt, we evaluate the return expression e to obtain the symbolic expression for the return value. We then check the return type of the function, where $\Gamma(\mathcal{E})$ denotes the inferred type for the symbolic expression. We update the return values set $\mathbb{R}$ with the symbolic return value and its corresponding symbolic condition. Finally, we update the break conditions π_{brk} recursively and the return condition π_{ret} to ensure that the following computations within this function are skipped.

5.2.4 Non Control Flow Statements. Handling non-control flow statements, specifically assertions and assignments, is not trivial, as they may introduce side effects to the program's state. We now discuss how ZinniaCC processes these statements based on the path condition in the symbolic state. We present the semantic rules for non-control flow statements in Figure 8.

Assertions. In typical programs, assertions are used to enforce certain constraints during program execution. In the context of zk-SNARKs circuits, assertions are used to apply additional arithmetic constraints. In Assert-Stmt, we obtain the path condition $\pi^{\mathbb{E}}$ for the current program execution point and the assertion enable condition π_{asrt} from the environment. We also evaluate the test expression e to obtain the symbolic expression $\mathcal{E}$ for this assertion statement. Hence, we construct a logical implication statement from these three symbolic expressions, namely $\pi^{\mathbb{E}} \wedge \pi_{\text{asrt}} \rightarrow \mathcal{E}$. This logical implication is simplified as $\neg(\pi^{\mathbb{E}} \wedge \pi_{\text{asrt}}) \vee \mathcal{E}$ and added to the set $\mathbb{C}$.

Assignments. Assignments are used to update the value of a variable, which updates the program's state. We separate the rules for assignment into three categories. Assign (New) is straightforward; as this variable is not defined before, we trivially evaluate the expression and update the variable table. However, when assigning to an existing variable, we must consider the path condition and the type consistency. In Assign (Existing, Mut), where we are able to alter the type of this variable,

we simply update the variable table with the new symbolic expression without considering the path condition. In Assign (Existing, Immut), where the type of this variable is immutable, we checks the type equality between the existing and the new value and use the selection function σ to select the correct value based on the path condition. Notably, the type mutable indicator θ here, computed by the function Θ, determines whether the type of a given variable can be modified. Further details are provided in Sec. 5.3, where we will also revisit those assign statements.

5.2.5 User-Defined Functions. Zinnia supports user-defined functions, which can be called within the program. In Figure 9, Function-Invoke handles function calls. We first look up the corresponding function in the functions set Λ to retrieve the function definition. Then, we create a new `FuncFrame` to store the function's return type, return values, and the corresponding symbolic conditions. We use $\pi^{\mathbb{E}} \wedge \pi^{\mathbb{E}}_{\text{asrt}}$ as the assertion enable condition for all assertions within the scope of this function. The types of the arguments are verified against the function's signature, and then the function body is evaluated. After that, we build a nested selection expressions using σ over all possible return values based on the return values and corresponding conditions stored in $\mathbb{R}$. Finally, we update the symbolic state to exit the function and provide the function's symbolic return value to the caller. Notably, Zinnia supports recursion and nested function calls, where we will raise an error if we cannot determine the recursion depth at compile time or it exceeds a predefined limit.

Function-Invoke

$$\mathbb{E}, \mathbb{C}, \mathbb{M} \vdash \{\text{new}(t_r)\} \Downarrow \mathbb{E}_0, \mathbb{C}_0, \mathbb{M}_0, \langle \mathcal{L}^{\text{ret}}, \mathcal{E}_0^{\text{ret}}, \dagger \rangle \quad \{id^{\text{func}} \mapsto (id_1 : t_1, id_2 : t_2, \ldots, id_M : t_M) \to t_r : s\} \in \Lambda^{\mathbb{E}}$$
$$\mathbb{E}_{i-1}, \mathbb{C}_{i-1}, \mathbb{M}_{i-1} \vdash \{e_i\} \Downarrow \mathbb{E}_i, \mathbb{C}_i, \mathbb{M}_i, \langle \mathcal{L}_i, \mathcal{E}_i, \theta_i \rangle, \text{where } i \in \{1, 2, \ldots, N\} \quad \forall i \in \{1, 2, \ldots, N\}, \Gamma(\mathcal{E}_i) = t_i$$
$$N = M \quad \text{ChipEnv}(\mathbb{E}_N, t_r, \pi^{\mathbb{E}} \wedge \pi^{\mathbb{E}}_{\text{asrt}}), \mathbb{C}_N, \mathbb{M}_N \vdash \{s\} \Downarrow \mathbb{E}_{\text{leave}}, \mathbb{C}_{\text{leave}}, \mathbb{M}_{\text{leave}}$$
$$\forall (\mathcal{E}_j^{\text{rc}}, \mathcal{E}_j^{\text{rv}}) \in \mathbb{R}^{\mathbb{E}} \text{ where } 1 \le j \le |\mathbb{R}^{\mathbb{E}}|, \mathcal{E}_j^{\text{ret}} = \sigma(\mathcal{E}_j^{\text{rc}}, \mathcal{E}_j^{\text{rv}}, \mathcal{E}_{j-1}^{\text{ret}}) \quad \wedge_{1 \le k \le |\mathbb{R}^{\mathbb{E}}|} \mathcal{E}_k^{\text{rc}} = \text{True}$$
$$\mathbb{C}_{\text{result}} = \mathbb{C}_{\text{leave}} \cup \{\forall (\mathcal{E}_j^{\text{rc}}, \mathcal{E}_j^{\text{rv}}) \in \mathbb{R}^{\mathbb{E}} \text{ where } 1 \le j \le |\mathbb{R}^{\mathbb{E}}|, \mathcal{E}_j^{\text{ret}} = \sigma(\mathcal{E}_j^{\text{rc}}, \mathcal{E}_j^{\text{rv}}, \mathcal{E}_{j-1}^{\text{ret}})\}$$
$$\mathbb{E}, \mathbb{C}, \mathbb{M} \vdash \{id^{\text{func}}(e_1, e_2, ..., e_N)\} \Downarrow \hat{\mathbb{E}}_{\text{leave}}, \mathbb{C}_{\text{result}}, \mathbb{M}_{\text{leave}}[\mathcal{L}^{\text{ret}} \mapsto \mathcal{E}^{\text{ret}}_{|\mathbb{R}^{\mathbb{E}}|}], \langle \mathcal{L}^{\text{ret}}, \mathcal{E}^{\text{ret}}_{|\mathbb{R}^{\mathbb{E}}|}, \dagger \rangle$$

Fig. 9. **Operational Semantics for User-Defined Functions.**

5.3 Type System

We now describe Zinnia's type system by introducing the built-in types, the static typing design and type inference, and how we ensure type consistency while offering variable shadowing flexibility.

5.3.1 Built-in Types. To offer developers the convenience of working with Zinnia using different types and making their programs more expressive, we support several built-in types, including integers (as field element), floats (as real numbers), and booleans as basic types, and lists, tuples, and tensors as composite types. Zinnia's composite types are constructed from basic types and can be nested to form complex data structures. We distinguish between composite types with different lengths or shapes, as knowing the lengths and shapes allows us to generate the correct number of arithmetic constraints needed for the corresponding computations.

5.3.2 Static Typing and Type Inference. Zinnia is designed as a statically typed language, which requires the type of each variable and expression to be known at compile time [29]. The rationale behind the static typing design is that arithmetic circuits are static in nature; hence, the type of each computation expressed in the program must also be statically known. Additionally, Zinnia allows writing programs without explicit type annotations for each variable declaration. While this design decision simplifies working with Zinnia, it also makes type inference a vital component in Zinnia's design. Type inference is the process of automatically inferring the type of each symbolic expression

during compilation [32]. In ZINNIA, the type inference process is denoted by the function $\Gamma(\mathcal{E})$, where $\mathcal{E}$ is the symbolic expression to be inferred. A comprehensive set of type inference rules has been developed, and while detailed, they are largely consistent with those found in standard programming language literature. For brevity, we omit their full presentation here.

5.3.3 Variable Shadowing and Type Consistency. Although ZINNIA is a statically typed language, we offer developers the flexibility of variable shadowing: the ability to change the type of a variable after it has been declared. This is drawn from the inspiration from languages like Rust, which allows developers to shadow variables by re-declaring them with the same name, not necessarily in the nested scope. This flexibility allows developers to write more expressive programs and work with ZINNIA with ease. Suppose that a variable is declared as an empty list, and later in the program, we fill this list with a `for` loop. Since lists of different lengths are considered as different types in ZINNIA, the type of this variable will change at each iteration. Without the flexibility to alter the type binding of this variable, developers would need to find workarounds, such as declaring a list with a fixed length, which would make the program less readable and harder to maintain.

This flexibility, however, introduces the challenge of maintaining type consistency. Since a variable's type may change after declaration, it can become dependent on path conditions, potentially causing inconsistencies. Thus, we must ensure type consistency across all execution paths. In Figure 10, the type of `ary` changes from a length-1 list to a length-6 list when `x` equals 0. The only distinction between the two programs is how `x` is defined. In the type-consistent program (left), `x` is a constant 0, ensuring the condition in `ary`'s assignment is known at compile time, maintaining type consistency. In the type-inconsistent program (right), `x` is an input integer, making the condition dependent on runtime input, leading to type inconsistency across execution paths and resulting in a type inference error during compilation.

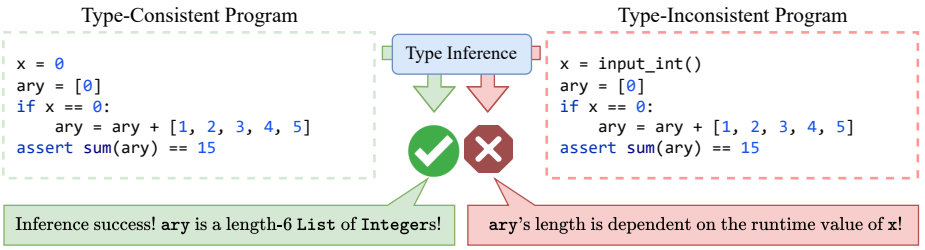

Fig. 10. **Example Illustrating Type-Consistent and Type-Inconsistent Programs.**

The challenge is to identify and report all type inconsistencies during type inference. To address this, we introduce type mutability indicators, θ, associated with each symbolic expression in memory but hidden from developers. θ can be † (mutable) or ‡ (immutable), tracking whether a variable's type can change.

Sec. 5.2.4 already discussed how ZINNIACC handles assignments w.r.t. the path condition. However, when assigning to an existing variable, we must also consider type consistency. With the help of type mutable indicators, we can tell whether this variable can be shadowed or not. In ASSIGN (EXISTING, MUT), the indicator is calculated as † (mutable), indicating that there are no other possible execution paths that might read the old value of this variable so we can change the type of this variable. In ASSIGN (EXISTING, IMMUT), the indicator is calculated as ‡ (immutable). In this case, the variable cannot be shadowed as there exists an execution path referencing this variable and we must use the selection expression σ to select the correct value for this variable based on the path condition.

Algorithm 1: The Type Mutable Indicator Function $\Theta_{id}^{\mathbb{E}}$

Input: variable identifier id, environment $\mathbb{E}$
Output: type mutable indicator $\theta \in \{\dagger, \ddagger\}$

if $\mathbb{E}$ ***is*** `IfFrame` **then**
  if $id \in \mathbb{E}.var\_table$ **then**
    return $\dagger$; // MUTABLE: this variable is defined in the current frame
  if $\bigwedge_{\mathcal{E} \in \pi_{if}^{\mathbb{E}}} \mathcal{E}$ **then**
    return $\Theta_{id}^{\hat{\mathbb{E}}}$; // INHERIT: the if condition is always true
  return $\ddagger$; // IMMUTABLE otherwise
if $\mathbb{E}$ ***is*** `LoopFrame` **then**
  if $\bigwedge_{\mathcal{E} \in \pi_{cont}^{\mathbb{E}} \cup \pi_{brk}^{\mathbb{E}}} \neg\mathcal{E}$ **then**
    if $id \in \mathbb{E}.var\_table$ **then**
      return $\dagger$; // MUTABLE: defined in the current frame and the iteration is guaranteed
    return $\Theta_{id}^{\hat{\mathbb{E}}}$; // INHERIT: this iteration is guaranteed to run
  return $\ddagger$; // IMMUTABLE: we cannot guarantee this iteration will run
if $id \in \mathbb{E}.var\_table$ **then**
  return $\dagger$; // MUTABLE: the variable is defined in the current frame
return $\Theta_{id}^{\hat{\mathbb{E}}}$; // INHERIT

Alg. 1 presents the type mutable indicator function $\Theta_{id}^{\mathbb{E}}$, which determines whether a variable's type can be altered. The core idea behind this algorithm is to check whether there exists any execution path referencing the value before the assignment. Generally, if a variable is defined within the current scope, it is consider mutable. Specially, if the path condition is not statically known, all variables are considered immutable. If the variable is defined in a parent scope, the mutable indicator is inherited from the parent scope only when the path condition is statically known to be true, otherwise it is considered immutable.

This variable shadowing design and the use of type mutable indicators ensures that the type of each variable is consistent across all possible execution paths and also provides the flexibility for developers to change the type of the variable when it does not lead to type inconsistency issues.

6 IMPLEMENTATION

We implement Zinnia in Python with about 20K LoC. Zinnia works as a source-to-source compiler where developers write high-level tensor operations in Python and compile them into optimized arithmetic circuits. Below, we introduce several key design considerations and implementation details.

6.1 Built-in Functions and Real Number Arithmetic

Table 2. Built-in functions and tensor operators in Zinnia.

Python Builtins	Functions	`range`, `any`, `all`, `sum`, `pow`, `abs`, ...
	Castings	`bool`, `int`, `float`, `list`, `tuple`
	Indicing & Slicing	`set_item`, `get_item`
Numerical Computations		`math.log`, `math.sqrt`, `math.exp`, ...
Tensor-oriented	Manipulations	`np.stack`, `np.concat`, `np.transpose`, ...
	Creations	`np.asarray`, `np.ones`, `np.zeros`, `np.eye`, ...
	Computations	`np.matmul`, `np.dot`, `np.argmax`, `np.argmin`, `np.add`, ...

To align with Python and NumPy programming conventions, we implement and optimize *nearly all* built-in functions and tensor operators from the Python and NumPy ecosystem in Zinnia (see Table 2). This design enables developers to seamlessly migrate their existing Python/NumPy code to Zinnia without needing to reimplement these functions.

Real-number arithmetic [26] is a cornerstone of modern computing systems, particularly for tensor operations. However, supporting real numbers in ZKP systems is challenging, as arithmetic circuits are inherently restricted to finite fields with integer-based operations. In zkVMs, real-number arithmetic relies on costly software emulation of floating-point operations using the RV32I instruction set. In other DSLs, such support is typically absent. Following the approach of popular ZKML frameworks [11, 16, 20, 35], we adopt a fixed-point representation in Zinnia to approximate real numbers. This method is significantly more efficient than emulation-based floating-point arithmetic in ZKP systems. As a result, Zinnia enables efficient real-number arithmetic operations at the cost of slight deviations from standard floating-point behavior. Having said that, to achieve behavior equivalent to standard real-number operations, we anticipate future work incorporating emulation techniques similar to those used in zkVMs.

6.2 IR, Optimizations, and Backends

As delineated in Sec. 5, Zinnia employs a symbolic execution approach to compile high-level programs into a set of computation statements, denoted as $\mathbb{C}$. To systematically encapsulate those computation statements, we design the Zinnia IR (ZinniaIR). ZinniaIR consists of a set of computational instructions, each representing a single constrained computation. Control flow instructions are excluded, as they are managed by the ZinniaCC. ZinniaIR is formulated in the SSA form, which enables the application of optimization and transformation techniques drawn from existing compiler literature. In our current implementation, Zinnia provides a suite of optimizations based on ZinniaIR to minimize the number of generated arithmetic constraints, including constant folding, dead code elimination, and common subexpression elimination [2].

Zinnia functions as a source-to-source compiler, translating program as inputs into sources for SNARK libraries and cDSLs. Our implementation primarily targets PLONK arithmetic circuits using the Halo2 backend while we also provide experimental support for Groth16 using Circom [5].

7 EVALUATION

We conduct an evaluation of Zinnia and aim to answer the following research questions:

- **RQ1: Expressiveness and Usability.** Is Zinnia expressive in implementing real-world zero-knowledge applications, and how does it compare to existing zkVM systems in terms of usability?
- **RQ2: Optimization Effectiveness.** Compared to SNARK-native libraries & low-level cDSLs, how do the optimizations implemented in Zinnia impact constraint count and performance?
- **RQ3: Superiority against zkVM.** Compared to zkVM systems with similar expressiveness and usability, how superior is Zinnia, as a SNARK-native system, in proving and verifying time?

To answer **RQ1**, we manually implement 25 programming tasks from various domains using Zinnia, Halo2, RISC0, and SP1, respectively, and compare the complexity of the code. Then, we further conduct a user study to understand the developers' workloads when using Zinnia and Cairo, a zkVM system known for its usability. To address **RQ2**, we compare it against the vanilla Halo2 system. Recall that Zinnia is built on top of Halo2, and the baseline comparison helps us understand the impact of the optimizations enabled by Zinnia. Finally, to address **RQ3**, we compare Zinnia against two zkVM systems, RISC0 and SP1, to understand the performance advantages of Zinnia's SNARK-native approach over zkVM architectures. Below, we first introduce the experimental setup and then present the results for each research question in turn.

7.1 Experimental Setup

Table 3. Programming tasks in the evaluation (# denotes the problem ID in the respective platform).

MLAlgo	LeetCode										DS-1000					Crypt
① Neuron; ② K-Means; ③ Linear-Regression	Array		DP		Graph		Math		Matrix		#296	#330	#387	#453	#501	① Poseidon Hash; ② Baby Jubjub ECC
	#204	#832	#740	#1137	#3112	#997	#492	#2125	#73	#2133	#309	#360	#418	#459	#510	

Datasets. We prepare 25 programming tasks (detailed in Table 3) from four sources: **MLAlgo**, **LeetCode**, **DS-1000** and **Crypt**. **MLAlgo** consists three machine learning tasks: Neuron, K-Means, and Linear Regression. **LeetCode (LC)** consists of ten programming tasks randomly selected from LeetCode [25] and covers five categories (Array, Dynamic Programming, Graph, Math, Matrix), with two difficulty levels (one easy, one medium) per category. **DS-1000** consists of ten data science tasks randomly sampled from DS-1000 [23], a corpus of 1000 data science tasks in Python. **Crypt** consists of two cryptographic tasks: Baby Jubjub ECC [6] and Poseidon Hash [18]. In summary, the above programming tasks cover a wide range of computational workloads that are representative in real-world ZKP application scenarios. Based on these tasks, we implement the same algorithms in Zinnia, Halo2, RISC0, and SP1, respectively, to support our evaluation below. The implementations are cross-verified to ensure correctness.

User Study. We recruit six participants with Python and Rust programming expertise and no prior experience with ZKP-related systems. The participants are divided into two groups: the Zinnia Group and the Cairo Group. As a baseline, Cairo is a zkVM system that compiles Rust programs into zkSNARK circuits at the cost of higher performance overhead. We refrain from using low-level frameworks like Halo2 in the user study as these systems come with a steep learning curve and are not likely for participants to understand within the study's timeframe. Participants are provided with introductory learning materials for their respective systems and are required to take a quiz to demonstrate their understanding and be eligible for the study. After passing the quiz, participants are asked to implement three elementary programming tasks in Zinnia and Cairo within 60 minutes, respectively. The tasks include ① testing whether a given number is prime, ② calculating staircase climbing combinations, and ③ determining path existence in a directed graph represented as a matrix. Task completion time and correctness are measured to evaluate usability. During the study, participants are allowed to use online resources to simulate real-world development scenarios.

Environment. All experiments are conducted on a server with dual Intel(R) Xeon(R) Gold 6444Y CPUs @ 3.60GHz and 256GB RAM. Multithreading (64 threads) is enabled and all baselines run with its default configurations.

7.2 RQ1: Expressiveness and Usability

To evaluate the expressiveness of Zinnia, we implement the 25 programming tasks from the datasets in Sec. 7.1 and compare the code complexity against Halo2, RISC0, and SP1. All methods are capable of implementing these tasks. We report the maximal control flows and the average LoC in Table 4. These tasks involve complex control flows with loops and conditionals, with the most complex case consisting of 5 nested loops, and multiple conditional statements. Programming such tasks are inherently challenging in low-level systems like Halo2, while Zinnia significantly simplifies the process. In particular, Zinnia delivers 2–3× shorter code than the baseline systems on average, with a peak reduction of over 6× for DS-1000 tasks. The rationale is multi-fold: first, Zinnia inherits Python's high-level syntax that simplifies the expression of complex control flows, compared to Rust in RISC0/SP1 or the low-level constraints in Halo2; second, Zinnia provides a

Table 4. Code implementations report.

Dataset		MLAlgo	LeetCode	DS-1000	Crypt
Max. Ctrl. Flows	Loops	5	5	4	0
	Brk/Cont	0	1	1	0
	Branches	1	3	1	0
Avg. LoC	ZINNIA	24.0	13.0	5.90	15.0
	Halo2	67.0	42.5	31.5	62.5
	SP1	72.0	48.0	36.7	57.0
	RISC0	67.0	39.1	31.7	48.5

Table 5. Task completion time and correctness.

Participant		Time (in minutes) T1	T2	T3	Correctness T1	T2	T3	Tot. Time
Zinnia Group	P1	15	8	18	✓	✓	✓	41
	P2	12	6	25	✓	✓	✓	43
	P3	5	7	10	✓	✓	✓	22
ZINNIA Avg.		10.67	7.00	17.67	100%	100%	100%	
Cairo Group	P4	5	13	45	✓	✓	✓	63
	P5	16	13	**60**	✓	✓	×	89
	P6	11	15	**60**	✓	✓	×	86
Cairo Avg.		10.67	10.33	55.00	100%	100%	33.3%	

rich set of built-in operators and tensor operations that streamline programming, reducing the need for manual circuit construction, which is particularly beneficial for data science tasks in DS-1000.

We further conducted a case study to demonstrate the expressiveness of ZINNIA, highlighting three representative examples in Figure 11 (with additional cases detailed in the Appendix Sec. A). Case A showcases ZINNIA's implementation of LC-#492, a programming problem requiring dynamic control flow to determine rectangle dimensions with minimal length-width difference, using `for` loops with conditional branches to break the loop upon finding the correct answer. Case B illustrates ZINNIA's implementation of linear regression by gradient descent, leveraging built-in tensor operations with slicing operations that mimic NumPy, such as `np.dot` and `np.sum`, while its native support for real-number arithmetic simplifies computation. Case C demonstrates a UDF for primality testing, enabling reusable components for repetitive tasks and supporting early termination via `return` statement. Notably, all cases showcase ZINNIA's static type system with type inference, where users annotate types for inputs while types for intermediate values are inferred automatically. These cases collectively highlight ZINNIA's expressiveness in handling dynamic control flow, tensor operations, real-number arithmetic, UDFs and a flexible type system.

A: Construct the Rectangle (LC #492)

```
@zk_circuit  # indicates a circuit
def construct_rect(
  area: int, e_l: int, e_w: int
):
  w = area
  for i in range(1, 1001):
    if area % i == 0:
      w = i
    if i * i >= area:
      break
  # [some code omitted for brevity]
  assert a_l == e_l and a_w == e_w
```

B: Train a Linear Regression (MLAlgo)

```
@zk_circuit  # indicates a circuit
def linreg(data: NDArray[float, 10, 3]):
  X, y = data[:, :-1], data[:, -1]
  weights = np.zeros(X.shape[1])
  bias, m = 0.0, float(X.shape[0])
  for _ in range(100):
    preds = np.dot(X, weights) + bias
    errs = preds - y
    dw = (1.0 / m) * np.dot(X.T, errs)
    db = (1.0 / m) * np.sum(errs)
    weights -= 0.02 * dw
    bias -= 0.02 * db
  # [following code omitted ...]
```

C: Primality Test UDF (User Study T1)

```
@zk_chip  # indicates a UDF
def is_prime(number: int) -> bool:
  assert 0 <= number <= 10000
  if number < 2:
    return False
  for i in range(2, 101):
    if i * i > number:
      return True
    if number % i == 0:
      return False
  return True
```

Fig. 11. **A Study on Representative Cases using ZINNIA.**

We further launch a user study to evaluate the usability of ZINNIA. We compare the completion time and correctness of the tasks between ZINNIA and Cairo. Here, Cairo serves as a baseline due to its established usability in the zkVM landscape. We report the program correctness and completion time for each participant in Table 5, and the correctness is determined by passing all test cases. Compared to Cairo, participants using ZINNIA exhibit a trend of faster task completion (p-value ≤ 0.05 with Mann-Whitney U Test), especially for Tasks 2 and 3. In terms of program correctness, most tasks were successfully implemented by our participants. However, Participants 5 and 6 from

the Cairo Group were unable to complete Task 3 within the allocated time. Feedback from the Cairo Group indicated that they found it challenging to work with Cairo's arrays, which only support appending and front-removal operations, as well as converting between field elements and usize in Rust, both of which contributed to their difficulty in completing Task 3.

Answer to RQ1: Zinnia is expressive and capable of implementing a wide range of ZKP applications with significantly shorter code than baseline systems. The user study further validates its usability with reduced task completion times and improved correctness compared to Cairo.

7.3 RQ2: Optimization Effectiveness

To understand the impact of our optimizations, we measure the following metrics: ❶ proving and verifying time, and ❷ constraint reductions. The constraint count serves as a proxy for circuit size and complexity, which directly impacts proving and verifying time. We collect these metrics from ten independent runs per task and implementation and report the average result. We first study the improvement over hand-crafted circuits in Halo2, and then perform an ablation study to understand the contribution of Zinnia's optimizations.

Figure 12 illustrates the constraint reductions achieved by Zinnia compared to Halo2 for each task along with the corresponding proving and verifying time. In terms of constraint reductions, Zinnia consistently outperforms Halo2 across various tasks. On average, Zinnia reduced the number of constraints by 19.3%, with a peak reduction of 98.7% for LC-#3112.

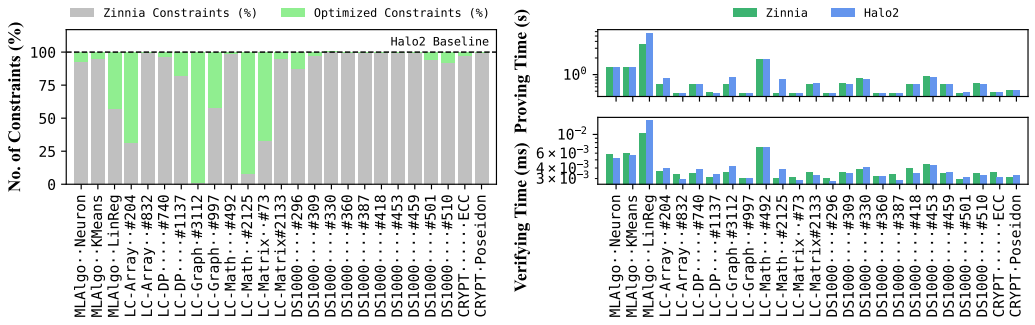

Fig. 12. **Constraint Reductions, Proving & Verifying Time for Zinnia and Hand-Crafted Circuits in Halo2.** On average, Zinnia reduces the constraints and proving time by 19.3% and 4.9%, respectively. The verifying time is nearly the same.

To elucidate the mechanisms underlying constraint reductions achieved by Zinnia's optimization, we conduct a detailed analysis of several test cases. In LC-#204, which implements a prime number counting algorithm, a significant reduction in constraints is observed (68.6%). This reduction stems from the constant folding optimization, which identifies and replaces redundant computations with precomputed values. Given that the number of primes below a fixed limit is a constant, it replaces the computationally intensive sieve algorithm with the precomputed result. Similarly, for LC-#3112, as an implementation of the Floyd-Warshall algorithm, the optimizer eliminates redundant computations by caching and reusing intermediate results. This effectively reduces the constraints by avoiding repeated calculations of shortest paths. A related optimization is observed in LC-#2125, where a brute-force laser counting algorithm is streamlined through the elimination of repeated calculations. In DS-1000#510, a data science task involving tensor operations, the optimizer addresses redundant zero checks. Specifically, it identifies and optimizes

repeated checks for zero rows and columns within the tensor. Across these benchmarks, Zinnia's optimizers consistently identifies and addresses optimization opportunities, including constant computation, redundant calculation elimination, and dead code removal, which collectively contribute to substantial reductions. We also investigate several cases where Zinnia's optimizations are less effective (e.g., `Neuron`, `K-Means`, `LC-#492`), typically involving complex control flows or data dependencies that lack specific patterns the optimizer can exploit, while for the remaining less effective cases, we identify that the manual implementations are already optimized, leaving little room for the optimizers to further reduce.

We also compare proving and verifying time against baseline system. In zk-SNARKs systems, these metrics are crucial for performance, where proving/verifying time denotes the time to generate a proof of this program, and the time to validate the proof, respectively. Given that proving time is typically orders of magnitude greater than verifying time and highly computationally intensive, it is often the more critical metric. Since constraint count serves as a proxy for circuit size and constraint complexity, Zinnia's constraint reductions directly influence these time. As demonstrated, Zinnia consistently achieves a lower proving time compared to the baseline across most tasks (4.9% on average). However, the verifying time are comparable, with both Zinnia and the baseline exhibiting very short durations (less than 5 milliseconds). Consequently, discerning significant differences in the verifying time is challenging, as they are likely overshadowed by other system-level factors.

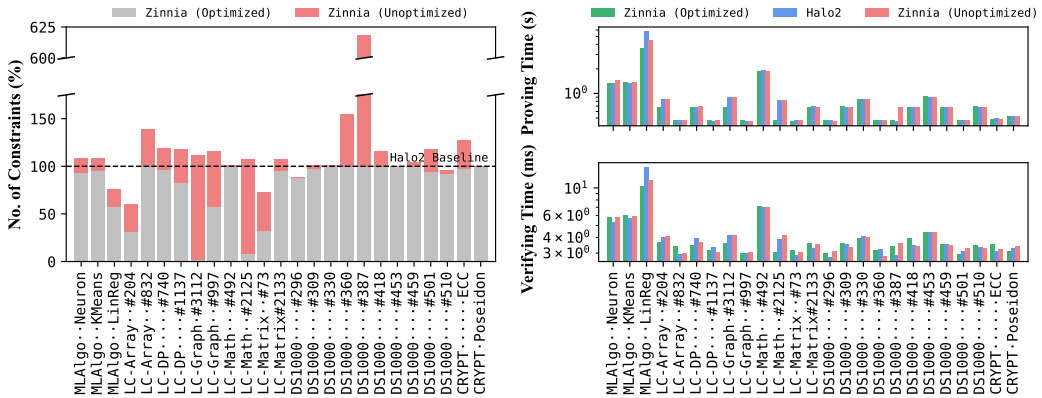

Fig. 13. **Ablation Study of Zinnia Optimizers.** The figure on the left shows the constraint reductions / increases while the figure on the right shows the proving and verifying time without optimizers.

To assess the contribution of Zinnia's optimizers to performance, we launch an ablation study. With ZinniaCC's API, we disable all optimizers and subsequently measure the effects on constraint count, proving time, and verification time. We report the results in Figure 13. As anticipated, deactivating the optimizers yields a considerable surge in constraints (46.0% on average), which, in turn, extends both proving and verification durations (8.5% and 1.9% on average, respectively). Notably, for the majority of tasks, the constraints exceeded our hand-crafted circuits in Halo2 (the horizontal dotted line in the left-hand plot). For `DS-1000#387`, the initial circuit contains 1384 constraints, which is 6.2× larger than the hand-crafted circuit in Halo2. After optimization, the circuit size is on par with the hand-crafted circuit. Likewise, for `LC-#3112`, the optimizations trim a considerable amount of unnecessary constraints (98.9%). This ablation study effectively highlights the indispensable role of Zinnia's optimizers in reducing constraints and improving overall performance.

Answer to RQ2: On a diverse set of ML, progamming, data science and cryptographic tasks, Zinnia effectively reduce constraints, and consequently, proving and verifying time, compared to manual circuit implementation in Halo2. The improvement is primarily attributed to Zinnia's powerful optimizations that eliminate redundant computations and streamline circuit structures.

7.4 RQ3: Superiority against zkVM

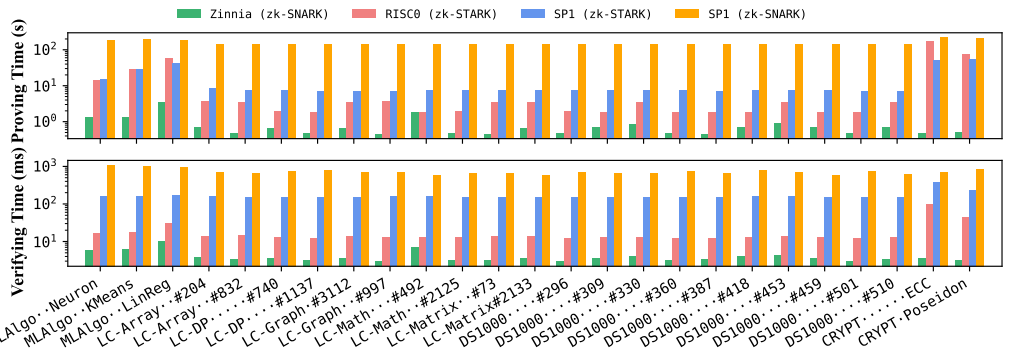

Fig. 14. **Comparison of Proving and Verifying Time for Zinnia against zkVM Solutions.** On average, Zinnia outperforms RISC0, SP1, and SP1's SNARK prover by 25.4×, 20.3×, and 245.4× in proving time and by 5.0×, 45.1×, and 191.7× in verifying time, respectively.

Following the experimental setup, we compare the proving and verifying time of Zinnia against RISC0, SP1, and SP1's SNARK prover. Since both RISC0 and SP1 employ a distinct arithmetic circuit model, we do not report the constraint count in this RQ. Instead, we focus on the performance advantages of Zinnia over tools with similar usability but different underlying architectures.

Before proceeding with the results, we recap that key differences exist between zkVM systems and Zinnia. In a nutshell, zkVM systems compile programs into CPU instructions and simulate CPU execution to generate proofs. To accommodate this computation model, zkVMs typically first generate a zk-STARK proof, which is then wrapped in a SNARK proof for compatibility with blockchains. In contrast, Zinnia directly compiles high-level programs into SNARK-native arithmetic circuits, which eliminates the need for CPU emulation and proof wrapping, while also providing strong usability.

Figure 14 compares the proving and verifying time of Zinnia, RISC0 (STARK), SP1 (STARK), and SP1's SNARK prover. The results show that zkVM solutions (RISC0, SP1, and SP1's SNARK prover) exhibit significantly longer proving and verifying time compared to Zinnia (over 25.4× and 5.0×, respectively). The reason for this disparity is three-fold. First, ZKP systems simulate a CPU to execute programs and prove the correctness of CPU state transitions. A CPU state incorporates a large number of registers, memory, program counters, etc., which necessitates a considerable amount of computation to generate proofs. In contrast, Zinnia directly compiles programs into arithmetic circuits, which are more amenable to optimization and avoids CPU emulation overhead. Second, ZKP systems operate on a finite field and only natively support integer arithmetic. As a result, these systems require considerable overhead to simulate real number arithmetic (floating-point) using integer arithmetic; while Zinnia alleviates this issue by directly compiling real number arithmetic into low-cost fixed-point arithmetic circuits. Third, wrapping zk-STARK proofs in zk-SNARKs introduces additional computational overhead, that further exacerbates the performance disparity.

Answer to RQ3: As a SNARK-native system, Zinnia outperforms zkVM solutions like RISC0 and SP1 that rely on CPU emulation for proof generation. Zinnia's direct compilation paradigm eliminates the need for CPU emulation and enables additional ZK-friendly optimizations to achieve superior performance.

8 DISCUSSION AND FUTURE WORK

Scalability. Zinnia employs symbolic execution, which is prone to classic challenges like memory aliasing and path explosion. We mitigate memory aliasing through multiplexing, which ensures *sound* management of symbolic memory access and updates. Path explosion, while a concern, is manageable in our research context for several reasons. First, Zinnia is tailored as a compiler for ZKP applications, where programs are typically small with limited execution paths. Second, arithmetic circuits inherently require encoding all possible execution paths, making path explosion unavoidable even in manually designed circuits without symbolic execution. Third, Zinnia mitigates this by pruning infeasible paths via constant propagation, as a best-effort optimization at compile time. Finally, aligned with prior works [5, 15], Zinnia achieves linear-time compilation complexity relative to circuit size, which ensures practical compiler efficiency for typical ZKP programs.

Zero-knowledge Program Analysis. A concurrent direction of research is the development of program analysis tools for ZKP programs/circuits. These tools aim to detect functional bugs [38] and security vulnerabilities [27, 28, 36], as well as to verify program properties [12, 24]. Zinnia mitigates potential bugs by abstracting the low-level circuit generation process, which reduces the likelihood of introducing errors during development time. However, in general, these tools are complementary to Zinnia and can be integrated to enhance the overall quality of ZKP programs.

Future Work. We anticipate several directions for future work to improve Zinnia. First, the current implementation of Zinnia relies on a transpilation paradigm, leveraging existing cDSLs and ZKP libraries for proof generation and verification. This dependency introduces limitations in achieving aggressive optimizations, as the abstractions provided by these underlying libraries can impede low-level control. Furthermore, reliance on external libraries exposes Zinnia to potential bugs or performance bottlenecks inherent in those dependencies. To mitigate these issues, future work will focus on developing Zinnia as a self-contained framework capable of native proof generation and verification, thereby enhancing reliability and potentially improving performance.

Second, the expressiveness of Zinnia will be expanded through the incorporation of a broader range of tensor operators, non-linear functions, and real number arithmetization schemes. This extension will enable Zinnia to support a wider array of applications, particularly those involving complex mathematical computations and machine learning models.

Third, the integration of constraint solvers into the compilation process will be explored. Currently, ZinniaCC performs symbolic execution, and then mainly employs constant propagation to identify compile-time constants. However, this method is limited in its ability to resolve complex expressions, resulting in suboptimal code generation and reduced expressiveness. By incorporating constraint solvers over symbolic expressions yielded by symbolic execution, ZinniaCC may perform more sophisticated static analysis, enabling the determination of a broader range of compile-time constants and thereby improving efficiency and versatility.

Finally, the programming language capabilities of Zinnia will be augmented to include features such as user-defined data types, string manipulation, and generics. These enhancements will leverage the power of symbolic execution to reason about program behavior, thereby increasing the expressiveness and applicability of Zinnia across diverse domains.

9 CONCLUSION

We introduced Zinnia, a ZKP programming framework designed to address the challenges of writing and optimizing ZKPs for machine learning and data science. Zinnia's high-level language, symbolic execution approach, and tensor-oriented optimizations enable high utility, expressiveness, and performance improvements over existing solutions. This work demonstrates the potential for Zinnia to accelerate the adoption of ZKPs across diverse applications.

REFERENCES

[1] Harold Abelson and Gerald Jay Sussman. 1996. *Structure and interpretation of computer programs*. The MIT Press.

[2] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. *Compilers principles, techniques & tools*. pearson Education.

[3] Nada Amin, John Burnham, François Garillot, Rosario Gennaro, Chhi'mèd Künzang, Daniel Rogozin, and Cameron Wong. 2023. LURK: Lambda, the ultimate recursive knowledge (experience report). *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 259–274.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. https://doi.org/10.1145/3182657

[5] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2022. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing* 20, 6 (2022), 4733–4751.

[6] Marta Bellés-Muñoz, Barry Whitehat, Jordi Baylina, Vanesa Daza, and Jose Luis Muñoz-Tapia. 2021. Twisted edwards elliptic curves for zero-knowledge circuits. *Mathematics* 9, 23 (2021), 3022.

[7] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive* (2018).

[8] Manuel Blum, Paul Feldman, and Silvio Micali. 1988. Non-interactive zero-knowledge and its applications. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*. 103–112.

[9] Sean Bowe, Jack Grigg, and Daira Hopwood. 2019. Recursive proof composition without a trusted setup. *Cryptology ePrint Archive* (2019).

[10] Darko Čapko, Srđan Vukmirović, and Nemanja Nedić. 2022. State of the art of zero-knowledge proofs in blockchain. In *2022 30th Telecommunications Forum (TELFOR)*. IEEE, 1–4.

[11] Bing-Jyue Chen, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang. 2024. Zkml: An optimizing system for ml inference in zero-knowledge proofs. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 560–574.

[12] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. 2021. Leo: A programming language for formally verified, zero-knowledge applications. *Cryptology ePrint Archive* (2021).

[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices* 46, 3 (2011), 265–278.

[14] Electric Coin Co. 2025. halo2. https://github.com/zcash/halo2.

[15] 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. https://doi.org/10.1109/Cybermatics_2018.2018.00199

[16] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. 2021. Zen: An optimizing compiler for verifiable, zero-knowledge neural network inferences. *Cryptology ePrint Archive* (2021).

[17] U. Fiege, A. Fiat, and A. Shamir. 1987. Zero knowledge proofs of identity. (1987). https://doi.org/10.1145/28395.28419

[18] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A new hash function for {Zero-Knowledge} proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 519–535.

[19] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 305–326.

[20] ZKonduit Inc. 2025. EZKL. https://ezkl.xyz/.

[21] J. C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19 (1976), 385–394. Issue 7. https://doi.org/10.1145/360248.360252

[22] Succinct Labs. 2025. SP1. https://github.com/succinctlabs/sp1.

[23] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: a natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 756,

27 pages.
[24] 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. https://doi.org/10.1109/SP54263.2024.00078
[25] LeetCode LLC. 2025. LeetCode. https://leetcode.com/.
[26] Jean-Michel Muller, Nicolas Brunie, Florent De Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of floating-point arithmetic*. Vol. 1. Springer.
[27] Trail of Bits. 2025. circomspect. https://github.com/trailofbits/circomspect.
[28] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. 2023. Automated detection of under-constrained circuits in zero-knowledge proofs. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1510–1532.
[29] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.
[30] Pologon. 2025. Polygon ZK-EVM. https://polygon.technology/polygon-zkevm.
[31] scipr lab. 2025. libsnark. https://github.com/scipr-lab/libsnark.
[32] Robert W Sebesta, Soumen Mukherjee, and Arup Kumar Bhattacharjee. 1999. *Concepts of programming languages*. Vol. 7. Addison-Wesley Reading, Massachusetts.
[33] StarkWare. 2025. Cairo. https://github.com/starkware-libs/cairo.
[34] Alan Mathison Turing et al. 1936. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math* 58, 345-363 (1936), 5.
[35] Suppakit Waiwitlikhit, Ion Stoica, Yi Sun, Tatsunori Hashimoto, and Daniel Kang. 2024. Trustless audits without revealing data or models. In *Proceedings of the 41st International Conference on Machine Learning*. 49808–49821.
[36] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. 2024. Practical Security Analysis of {Zero-Knowledge} Proof Circuits. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1471–1487.
[37] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. 2021. Mystique: Efficient conversions for {Zero-Knowledge} proofs with applications to machine learning. In *30th USENIX Security Symposium (USENIX Security 21)*. 501–518.
[38] Dongwei Xiao, Zhibo Liu, Yiteng Peng, and Shuai Wang. 2025. Mtzk: Testing and exploring bugs in zero-knowledge (zk) compilers. In *NDSS*.
[39] RISC Zero. 2025. RISC Zero. https://github.com/risc0/risc0.
[40] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. 2020. Zero knowledge proofs for decision tree predictions and accuracy. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2039–2053.

A CASE STUDY

In this supplementary section, we present more case studies complementing the main paper to demonstrate the expressiveness of ZINNIA. While this showcase is not exhaustive, it highlights key examples of its capabilities. We provide a brief overview of our case studies in Table 6.

Table 6. Corresponding Case Studies for each Language Feature.

Control Flows	UDFs	Tensor Operations	Numerical Comp.	Slice & Index	Type Inference	Var. Shadowing
A.1	A.1, A.7	A.2, A.3, A.4, A.5	A.2, A.4	A.5, A.6	*All Cases*	A.6

A.1 Case 1: Test Prime Number (User Study Task 1)

In this case study we implemented a ZKP circuit using ZINNIA for integer primality checking. Figure 15 presents a possible implementation. The code defines a function, `is_prime`, annotated with `@zk_circuit`, signifying its compilation into a ZKP circuit. This function takes an integer and a boolean, representing the number to be tested and the expected primality result, respectively. Assuming the input is bounded by 10,000, the code initially asserts that numbers less than 2 are not prime. It then iterates from 2 to 100, checking for divisibility. To ensure computational accuracy, a `break` statement terminates the loop when the square of the iterator exceeds the input. This case

```python
@zk_circuit
def is_prime(number: int, prime: bool):
    assert 0 <= number <= 10000
    if number < 2:
        assert not prime
    else:
        result = True

        for i in range(2, 101):
            if i * i > number:
                break
            if number % i == 0:
                result = False

        assert prime == result
```

Fig. 15. **Zinnia Code for Case Study 1.**

```python
@zk_chip
def is_prime(number: int) -> bool:
    assert 0 <= number <= 10000
    if number < 2:
        return False
    for i in range(2, 101):
        if i * i > number:
            return True
        if number % i == 0:
            return False
    return True

@zk_circuit
def test_is_prime(number: int, prime: bool):
    assert is_prime(number) == prime
```

Fig. 16. **Alternative Code using UDF.**

demonstrates Zinnia's ability to handle complex control flows which is not common in current DSLs and libraries.

To enhance modularity and code reuse, Figure 16 demonstrates an alternative approach using UDFs which we've already discussed in the main paper. The function `is_prime` encapsulates the primality check, returning a boolean. The `@zk_chip` annotation designates this function as a UDF supported in Zinnia. Similar to traditional programming languages, Zinnia allows users to return values at any point and immediately exit the function, which is not well-supported in existing DSLs. This feature allows users to write programs with better program structure, readability, and maintainability.

A.2 Case 2: Linear Regression (`MLAlgo-LinReg`)

```python
@zk_circuit
def linear_regression(
    training_x: NDArray[float, 10, 2],
    training_y: NDArray[float, 10],
    testing_x: NDArray[float, 2, 2],
    testing_y: NDArray[float, 2]
):
    weights = np.zeros((training_x.shape[1], ))
    bias, m = 0, float(len(training_y))
    for _ in range(100):
        predictions = np.dot(training_x, weights) + bias
        errors = predictions - training_y
        dw = (1.0 / m) * np.dot(training_x.T, errors)
        db = (1.0 / m) * np.sum(errors)
        weights -= 0.02 * dw
        bias -= 0.02 * db
    test_predictions = np.dot(testing_x, weights) + bias
    test_diff = test_predictions - testing_y
    test_error = np.sum(test_diff * test_diff) / len(testing_y)
    assert test_error <= 1.0
```

Fig. 17. **Zinnia Code for Case Study 2.**

```python
@zk_circuit
def extract_patches(
    a: NDArray[int, 4, 4],
    result: NDArray[int, 4, 2, 2]
):
    assert a.reshape(
        (a.shape[0]//2, 2, a.shape[1]//2, 2)
    ).transpose(
        (0, 2, 1, 3)
    ).reshape((4, 2, 2)) == result
```

Fig. 18. **Zinnia Code for Case Study 3.**

```python
@zk_circuit
def calc_entropy(
    X: NDArray[float, 10], ans: float
):
    assert sum(X) == 1.0
    assert ans == np.sum(-X @ np.log2(X.T))
```

Fig. 19. **Zinnia Code for Case Study 4.**

This case study highlights Zinnia's proficiency in numerical computations and tensor operations. We demonstrate a linear regression model implemented using gradient descent. Figure 17 shows

the implementation in Zinnia. The code initializes weights and bias, then iteratively updates them based on the gradient descent formula. The error computation is also included. With the built-in real number support and tensor operations such as mat-mul, Zinnia simplifies the implementation of this machine learning task, making it more accessible to users with limited zero-knowledge programming experience.

A.3 Case 3: Extract Blocks or Patches (DS-1000#387)

This case study showcases Zinnia's tensor manipulation capabilities. The task involves extracting image blocks or patches (sampled from DS-1000). Figure 18 demonstrates this using Zinnia's built-in tensor operations, specifically `np.reshape` and `np.transpose`. By providing these operators, Zinnia simplifies the implementation of complex multi-dimensional array manipulations, enhancing both user development ease and code expressiveness.

A.4 Case 4: Calculate Shannon Entropy

In this case we demonstrate Zinnia's application in calculating the Shannon entropy of a normalized probability distribution. Figure 19 presents the corresponding code. The function `calc_entropy`, annotated with `@zk_circuit`, accepts a list of floats representing the probability distribution and the expected entropy as inputs. It utilizes the tensor-oriented operator `np.sum` and the non-linear function `np.log2` to compute the entropy.

A.5 Case 5: Data Binning (DS-1000 #418)

This case study showcases Zinnia's tensor operations and slice/index capabilities. The task is to bin a 2-dim array into equal partitions of a given length (3) and then calculate the mean of each of those bins. Figure 20 demonstrates the implementation. In order to achieve this, the code uses combinations of tensor slicing operations combined with `np.reshape` to manipulate tensors. This case demonstrates Zinnia's ability to handle complex tensor operations and slice/index operations.

```
@zk_circuit
def data_binning(data: NDArray[float, 2, 5], result: NDArray[float, 2, 1]):
    new_data = data[:, ::-1]
    bin_data_mean = new_data[:, :(data.shape[1] // 3) * 3].reshape(
        (data.shape[0], 1, 3)).sum(axis=-1) / 3
    assert result == bin_data_mean
```

Fig. 20. **Zinnia Code for Case Study 5.**

A.6 Case 6: Remove All Zero Rows and Columns (DS-1000#510)

This case study demonstrates Zinnia's variable shadowing feature. The task is to remove all zero rows and columns from a 2-dim array. Figure 21 shows the implementation, where we append non-zero information to an empty list using a loop. Recall that lists of different length are treated as different types in Zinnia, with Zinnia's variable shadowing feature, users essentially "redefine" the original variable `zero_rows` each iteration with new type: a list whose length is 1 more. This feature allows users to change the type of variables defined in the outer or same scope, making code implementation more flexible.

A.7 Case 7: Point Addition on Baby Jubjub ECC (Crypt-BabyJubjubECC)

This case showed Zinnia's ability to handle elliptic curve operations over finite field, specifically point addition on Baby Jubjub elliptic curve. The code in Figure 22 shows a UDF `baby_add` that

```python
@zk_circuit
def remove_0_rows_cols(
    input: NDArray[int, 5, 6],
    result: NDArray[int, 3, 4]
):
    zero_rows = []
    zero_cols = []
    for i in range(5):
        zero_rows += [all(input[i, :] == 0)]
    for i in range(6):
        zero_cols += [all(input[:, i] == 0)]
    # [The following code omitted for brevity...]
```

Fig. 21. **Zinnia Code for Case Study 6.**

```python
@zk_chip
def baby_add(
    x1: int, y1: int, x2: int, y2: int
) -> Tuple[int, int]:
    a = 168700
    d = 168696
    beta = x1 * y2
    gamma = y1 * x2
    delta = (-a * x1 + y1) * (x2 + y2)
    tau = beta * gamma
    x = (beta + gamma) * math.inv(1 + d * tau)
    y = (delta + a * beta - gamma) * math.inv(1 - d * tau)
    return x, y
```

Fig. 22. **Zinnia Code for Case Study 7.**

takes two points and returns a new point. To facilitate finite field arithmetic, Zinnia provides `math.inv` to compute the inverse of a number on the field. This case demonstrates Zinnia's ability to handle advanced cryptographic operations common in ZKP applications.