A zk-SNARK Programming Framework for Verifiable Data Analytics

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

zk-SNARKs enable succinct, efficiently verifiable proofs of correct computation over private data, but require the computation to be compiled into a *statically defined and bounded* arithmetic circuit. This requirement creates a multidimensional mismatch with data-analytic programming, which consists largely of array programs: (a) array operations and resulting shapes are dynamic; (b) arrays are accessed at data-dependent positions; (c) control flows are data-dependent. Existing zk-SNARK DSLs enforce this requirement using conservative syntactic restrictions that reject many bounded programs, or leave the programmer to bound and encode these by hand, while zkVMs recover expressiveness at substantially higher proving cost.

We present Zinnia, a zk-SNARK programming framework for analytics in which programs are written in a subset of Python with NumPy. Zinnia's compiler uses SMT-backed symbolic execution over array types with bounds to derive properties of the program's values that hold on every input, which we call *facts*. The facts bound every data-dependent shape and loop, which gives each array its capacity in the circuit. Built atop this compiler, Zinnia provides bounded-dynamic array and arithmetic operations for analytics. The same facts guide how these operators are lowered: for each operator, Zinnia considers known circuit constructions, whose cost depends on what the facts guarantee. Unlike a program, a circuit has no order in which a check protects later uses, so if a construction needs a fact, another constraint in the circuit must enforce it. A construction chosen for one operator may enforce a fact that another relies on. Zinnia therefore chooses the constructions of all operators together, seeking to minimize estimated circuit cost. Our evaluation on unmodified programs from three NumPy corpora shows that the circuits Zinnia compiles are never larger than optimized hand-written ones in existing DSLs, and that Zinnia achieves over 30× faster proving than zkVM-based approaches by geometric mean.

1 Introduction

Data analytics underpins modern knowledge and decision systems, enabling the ingestion, transformation, aggregation, and modeling of data [19, 78]. When the underlying data must remain confidential, verifying the correctness of analytic results becomes a fundamental challenge. Zero-Knowledge Proofs (ZKPs) [15] offer a principled solution: as shown in Figure 1, a *prover* can convince a *verifier* that a computation was executed correctly on private data, without revealing the data itself. Among ZKP protocols, zk-SNARKs [12, 17, 36] are particularly attractive for analytics workloads because they yield non-interactive, constant-time-verifiable proofs. Conceptually (Figure 2), the prover compiles the computation into an arithmetic circuit, produces both the output and an auxiliary witness, and generates a succinct proof of correct execution.

*Corresponding authors.

Authors' Contact Information: Zhantong Xue, Hong Kong University of Science and Technology, Hong Kong, China, zxueai@cse.ust.hk; Pingchuan Ma, Zhejiang University of Technology, Hangzhou, China, pma@zjut.edu.cn; Zhaoyu Wang, Hong Kong University of Science and Technology, Hong Kong, China, zwangjz@cse.ust.hk; Yuguang Zhou, Hong Kong University of Science and Technology, Hong Kong, China, yzhougv@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.

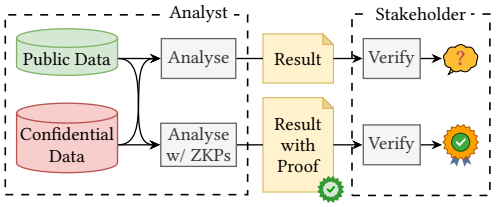


Fig. 1. **Private data analytics with ZKPs.**

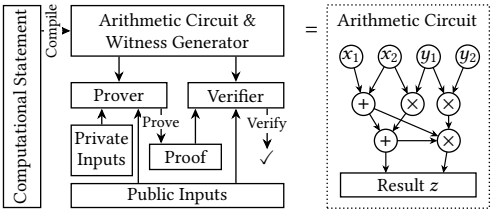


Fig. 2. **The zk-SNARK workflow.**

The Mismatch. At the heart of every zk-SNARK lies an arithmetic circuit: a fixed graph of additions and multiplications over a finite field [20]. This graph must be *statically defined and bounded*, i.e., its topology is determined entirely at compile time, independent of runtime inputs [9, 20]. We call this the *fixed-structure* requirement. While this fixed structure is essential for soundness and succinctness, it conflicts with how analytic programs are written.

Data analytics programming spans diverse paradigms, from relational query processing [35] and statistical computation [43] to large-scale machine learning and dataflow frameworks [1]. Despite their diversity, a large subset of analytics programs operates on array-oriented programming [44] that transforms, manipulates, aggregates, and reduces data stored in tabular or multidimensional array formats [40, 62]. Rather than relying on intricate control flow, they exploit and compose vectorized operations, allowing complex analyses to be expressed through high-level array primitives and operator pipelines [2]. Our work focuses on this prevalent class of data-analytic programs, which we call array programs: they compute by applying operators to whole arrays. Figure 3(a) shows an example which crops an image to the bounding box of its nonzero pixels. In this example, the fixed-structure requirement creates a *multidimensional mismatch* in shapes, accesses and control flow.

❶ **Shapes.** Filtering and slicing operations produce intermediate arrays whose dimensions are data-dependent. In this example, `rows` has one entry per nonzero row, so its length, and the shape of the output, depend on the pixel values.

❷ **Accesses.** Indexing, masking and sorting read or place elements at positions computed from the data. These cannot be directly mapped to a fixed circuit. In this example, the slice reads `im` at positions computed from `rows` and `cols`.

❸ **Control.** Branches and loops depend on the data. In this example, the `if` branches on a length.

A fixed circuit must provide for every input. A data-dependent shape needs room for the largest possible array, a *capacity*. A data-dependent access needs a *construction*, a way of expressing it as constraints, such as selecting the element with a one-hot vector or looking it up in a table. Data-dependent control needs both sides of every branch and every loop iteration up to a bound. Together, these choices determine the cost of every proof.

Writing fixed circuits by hand. Choosing capacities, constructions and bounds by hand is easy to get wrong. As a result, constructing such proofs remains difficult for data scientists and engineers, who rarely possess expertise in cryptography or low-level circuit design. Figure 3(b) shows that the 11-line example becomes a 139-line hand-written circuit. In Python-style analytics code, its slices and axis-wise reductions are first-class array operations. In Noir, however, array dimensions must be fixed at compile time: the developer must instead pad results to obtain a single circuit-compatible representation. The output is padded to 5×6 (❶), each read becomes a one-hot sum over all candidate positions (❷), and the branch is folded into arithmetic (❸). Each step requires knowing which constructions exist, what they cost and what they assume. A poorly chosen construction

(a) NumPy

```python
import numpy as np

def crop(im):
    mask = im == 0
    rows = np.flatnonzero((~mask).sum(axis=1))
    cols = np.flatnonzero((~mask).sum(axis=0))
    if rows.shape[0] == 0:
        result = np.array([])
    else:
        result = im[rows.min():rows.max()+1,
                    cols.min():cols.max()+1]
    return result
```

(b) Hand-written circuit (139 lines), excerpts

❶ output padded to 5×6, shape returned separately

```
fn main(im: [Field; R * C]) -> pub [Field; 2 + R * C] {
    // ...
    let mut out: [Field; 2 + R * C] = [0; 2 + R * C];
    out[0] = height;
    out[1] = width;
```

❷ each read is a sum over all positions

```
let mut x: Field = 0;
for d in 0..R - u {
    x += top[d] * src[(u + d) * C + j];
}
rows[u * C + j] = x;
```

❸ the `if` becomes arithmetic

```
let mut len: Field = (N as Field) * (1 + lead[N - 1]);
for i in 0..N {
    len -= lead[i] + after[i];
}
```

(c) Proving time

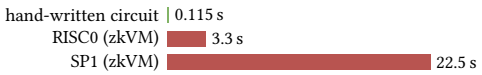


Fig. 3. **Cropping an image to the bounding box of its nonzero pixels** (DS-1000 problem 507). (a) NumPy code. (b) Excerpts of a hand-written circuit, in which ❶–❸ mark the lines of (a) they implement. (c) Proving time.

makes every proof slower, and a missing constraint lets a dishonest prover produce a wrong result with a valid proof, a common source of vulnerabilities [67, 90].

Existing approaches. DSLs such as Noir [64] and Circom [7] leave this work to the programmer, as in Figure 3(b). Prior work addresses narrow slices of this mismatch. CirC [65] applies SMT-backed reasoning to establish control-flow boundedness, but does not address the pervasive dynamism of array-programming primitives: it leaves capacities to the programmer and lowers each access by a fixed rule. Domain-specific systems for machine learning and SQL choose efficient constructions only for a fixed set of operators [18, 29, 58]. Proof folding schemes and recursive SNARKs [11, 17, 53] support Incrementally Verifiable Computation for unbounded loops, but incur cryptographic overhead at each folding step. Emulation-based approaches such as zkVMs [73, 81, 83] sidestep the mismatch by proving the execution of instructions, but this generality comes with order-of-magnitude proving overhead. In a zkVM, the developer can write this example with standard loops, slices, and assertions. However, proving proceeds over instruction-level execution traces (register, memory, and control-state transitions), so each high-level array operation expands into many low-level steps, yielding substantial overhead. Moreover, to preserve execution-length obliviousness, many of these systems typically commit to a fixed-size execution trace and pad the execution trace to the bound, which can substantially exceed the actual running length of many data-dependent programs and further amplify proving cost [9, 74]. For this example, proving takes 29–196× longer than for the hand-written circuit, as shown in Figure 3(c). This example thus shows that simple analytics tasks either require rigid circuit-friendly rewrites (in DSLs) or incur substantial emulation overhead (in zkVMs). Yet many such programs are *safe*: they are statically bounded despite their data-dependent appearance. Structurally, most array-oriented analytics programs can be modeled as finite dataflow or computation graphs, where operators form directed acyclic graphs that process bounded datasets [3, 70, 85]. A natural question arises: can we build a programming framework that systematically resolves the mismatch between zk-SNARKs' fixed circuit model and array-oriented data-analytic programming, recovering lost expressiveness without sacrificing circuit performance?

Observation. To meet the fixed-structure requirement, the author of a circuit reasons about what the program does on every possible input. The hand-written circuit of Figure 3(b) relies on such properties. `rows` and `cols` have at most five and six entries, so the output fits in a 5×6 capacity (❶). The box lies inside the image, so the output can be read by shifting whole rows and columns (❷). `rows` is empty exactly when every row is zero, which the running products already compute, so the branch becomes arithmetic (❸). We call such properties *facts*: claims about the program's values that hold on every input, in the sense of the dataflow facts and invariants of program analysis [23, 49]. Facts determine the capacities and constructions of the circuit, and through both its cost. They also cross operator boundaries: the facts above follow from the lines that find the box and are used by the branch and the slice. Facts, however, do not carry over to the circuit as they are. In a program, a check that has run protects every later use of a value. A circuit has no such order, since the prover chooses all values at once, so if a construction needs a fact, another constraint must enforce it. A fact that one construction enforces is then free for the others, while two constructions that rely on each other's facts check neither. The constructions of all operators must therefore be chosen together.

This paper. We show that a compiler can derive and use such facts automatically, and that doing so closes the gap between what analytics developers write and what zk-SNARK circuits require: array programs with data-dependent shapes, accesses and control compile to circuits as small as optimized hand-written ones. We present Zinnia, a zk-SNARK programming framework that embodies this insight. Zinnia programs are written in a subset of Python and compiled via an SMT-backed symbolic-execution engine into fixed arithmetic circuits. Built atop this compiler, Zinnia provides a library of *circuit-native* analytics primitives, including array permutation, reduction, broadcasting, slicing, indexing, and filtering, as well as real-valued arithmetic and nonlinear functions. To derive facts, we execute the program symbolically over its declared input domain (Sec. 4) and prove bounds on loop counts, array capacities and index ranges with an SMT solver. To choose constructions, we keep each array operator whole until the facts about its operands are known (Sec. 5). We then select a low-cost set of constructions, such as one-hot selection, lookup tables, sorting networks and checks with a random challenge from the verifier [32], in which every fact relied on is enforced by a selected construction, without circular reasoning. The correctness of the individual constructions then composes into the correctness of the whole circuit.
We summarize our contributions:

- **Problem Characterization.** We identify and articulate a *multidimensional mismatch* between zk-SNARK's fixed-structure requirement and array programs. Beyond loop unrolling, this mismatch encompasses control flow with data-dependent bounds, array operations that are dynamic and shape-changing, and accesses at data-dependent positions. We show that prior approaches address only narrow slices of this mismatch (Sec.2).
- **A Compiler from Array Programs to Fixed Circuits.** It lowers array operators by choosing constructions together with the facts they rely on and enforce, with soundness composed from per-construction arguments (Sec. 5). The facts come from an SMT-backed, whole-program pipeline that infers, via symbolic execution, the capacities, loop bounds and index ranges that zk-SNARK DSLs leave to the programmer (Sec. 4).
- **Comprehensive Evaluation.** Zinnia accepts 214 of 327 programs from three NumPy corpora. On the 172 compared, the compiled circuits are smaller on average than optimized hand-written ones, less than half the size of direct translations on Halo2, and proved 34–125× faster than in zkVMs, demonstrating *zero abstraction cost*: the improved expressiveness does not come at the expense of circuit performance (Sec.8).

2 Background

2.1 Zero-Knowledge Proofs, zk-SNARKs and Arithmetic Circuits

A Zero-Knowledge Proof is a protocol where a prover convincingly demonstrates to a verifier that a public statement s is true, while revealing nothing beyond the fact that it is true [15]. The prover knows a private *witness* w that makes s true. When s states that running a program on some private input gives the public output y, the witness holds the input and the intermediate values of the run. The protocol consists of three algorithms executed on common public parameters:

- **Setup.** On input the security parameter λ and the circuit C that checks the statement, $\mathsf{Setup}(\lambda, C)$ outputs a proving key pk and a verification key vk.
- **Proving.** Using pk and the witness w, the prover runs $\mathsf{Prove}(\mathsf{pk}, s, w)$ to generate a proof π.
- **Verification.** The verifier applies $\mathsf{Verify}(\mathsf{vk}, s, \pi)$, which runs in time at most polylogarithmic in the size of C. If the output is 1, the verifier is convinced that s is valid.

A ZKP system must satisfy three properties: *Completeness* means that if s is true, then an honest prover who knows w convinces the verifier to accept except with negligible probability. *Soundness* means that if s is false, then even a malicious prover cannot make the verifier accept except with negligible probability. *Zero-knowledge* means that whatever the verifier sees during the protocol can be simulated without access to w, so it learns nothing beyond the fact that s is true.

SNARKs are a family of ZKPs with small proof sizes and alleviate the need for interaction between the prover and verifier, which makes them particularly suitable for practical applications. SNARKs require expressing the statement s as a statically defined and bounded arithmetic circuit C under certain arithmetic systems (e.g., R1CS [9], PLONK [33]). The private witness w is then an assignment to C's private inputs and internal wires, and s is true if and only if some w satisfies C. Since the keys are derived from C once and C is public, its structure cannot depend on the private input. Since the prover chooses w, a compiler may rely on a property of the program only if a constraint of C enforces it.

Cost. Proving time grows with the size of the circuit, measured in *cells* or *gates* depending on the proof system. Arithmetic circuits operate over a large prime field $\mathbb{F}_p$, in which only the operations of addition and multiplication are allowed. Circuit designers have developed several ways to make an operation cheaper than a direct translation of its steps. Because the prover supplies every value, a circuit can check a result instead of computing it. For example, an integer division can take its quotient q and remainder r from the prover and check $a = q \cdot d + r$ and $0 \le r < d$, instead of performing long division. Lookup arguments replace bit-by-bit range checks and comparisons with a membership test in a precomputed table [71]. Some proof systems also let a circuit use random values chosen by the verifier, which make certain checks cheaper still, such as checking that one list is a rearrangement of another [33] or that $M = AB$ for matrices [32], at the price of a small, bounded probability of error.

2.2 Arrays in a Fixed Circuit

A circuit has neither memory nor control flow, so an array program must be mapped onto a fixed set of variables and constraints. Standard techniques exist for each way it depends on the data.

An array whose shape depends on the data is given a capacity, an upper bound on its size that is known at compile time, together with its runtime shape. Positions beyond the runtime shape hold padding, which must not affect the result.

Reading $x[i]$ at a private index i is harder, since the fixed-structure requirement is a poor match for programs that require *random access* on arrays: a circuit cannot select a variable at runtime. The read must instead be built from constraints, and many constructions are known for it. The

Table 1. **How existing systems handle data-dependent shapes, accesses and control flow.** "Programmer": the programmer writes it in the program; "generic": the construction is used regardless of the program.

	Data-dependent			Proving
	shapes	accesses	control	cost
Constraint-based DSLs	programmer	programmer	programmer	low
Program-based DSLs (Noir)	programmer	fixed rule	programmer	low
zkVMs (SP1, RISC0)	not needed	generic memory check	not needed	high
zkML / zkDB systems	fixed or given bounds	per operator	not applicable	low
Circuit compilers (CirC)	programmer	fixed rule	inferred	low
Zinnia	inferred	jointly chosen by cost	inferred	low

simplest selects the element with a one-hot vector, computing $\sum_j (j = i) \cdot x[j]$. This approach scales linearly in the length of x for every read [68], is sound, and is efficient when x is small. A lookup argument [71] is cheaper when x is a fixed table, and memory-checking arguments [14, 66, 88] handle arrays that are both read and written. Instead of hard-wiring a separate copy constraint for every possible address, the prover supplies a *memory trace*, a sequence of read and write events annotated with an address, a value, and an implicit time order. The circuit then checks a *RAM consistency argument*: each read of address a must return the value written by the most recent preceding write to a (or an agreed initial value if no write exists). A common enforcement strategy relates the unsorted trace to an auxiliary trace sorted by (address, time), checks local adjacency constraints in the sorted order (values persist across consecutive entries with the same address and change only at writes), and uses a permutation check to ensure the two traces contain the same events. This yields composable constraints for dynamic indexing while keeping the circuit statically defined and bounded by the trace length. Rearrangements can be realized with routing or sorting networks [88, 89]. Whole operations admit further constructions. A sort, for instance, can be verified with fewer constraints than computing it, by checking that the output is a rearrangement of the input and is in order [99]. Data-dependent control, finally, is handled by unrolling each loop up to a bound and selecting between both sides of each branch [68, 88].

These constructions differ in cost by large factors, and which of them are valid depends on what is known about the program's values. Reading a constant array takes a single table lookup, whereas reading an array that depends on the input selects from all its elements or shares a table among all reads. Because NumPy wraps negative indices and rejects out-of-range ones, a read must also check and adjust i, unless i is known to lie in range, as it does when $i = \texttt{np.argmax}(y)$ for an array y of the length of x. Choosing a construction well therefore requires facts about the program's values, which often come from elsewhere in the program.

2.3 Existing ZKP Solutions

We review existing ZKP solutions and discuss their limitations in the context of data analytics. We categorize them into five main classes: declarative libraries and constraint-based DSLs, program-based DSLs, zkVMs, domain-specific zkDB and zkML systems, and circuit compilers. We summarize their conceptual differences and trade-offs in Table 1.

Declarative Libraries and *Constraint-based DSLs* such as Circom [7], Halo2 [84], and LibSNARK [76], expose APIs for direct arithmetic-circuit construction. They represent the lowest level of abstraction, offering maximal flexibility but demanding detailed, manual control over gates and constraints. For data-analytic workloads, this model lacks abstractions for complex data structures or control flow. Users must manually decompose high-level operations into granular gate-level logic.

Program-based DSLs like ZoKrates [27], Leo [21], and Noir [64] allow developers to write structured, imperative programs that are automatically compiled into arithmetic circuits. They abstract away much of the low-level constraint management while maintaining proof efficiency. As they must still conform to the fixed-structure requirement, they reject programs whose loop bounds depend on input data, even when such loops are statically bounded and can be safely mapped to a fixed circuit topology. This restriction forces developers to rewrite natural data-dependent logic into rigid, fixed-length loops with masking or flag variables. They also require capacities to be written in the program, and they lower each data-dependent access by a fixed rule.

Zero-knowledge Virtual Machines (zkVMs) such as SP1 [83], Cairo [81], and RISC0 [73], emulate CPU instruction sets to support arbitrary control flow. Developers can write standard programs (e.g., in Rust), and the zkVM captures their execution trace as a sequence of state transitions verified within a circuit. This approach restores full programmability but at the cost of performance: instruction-level emulation produces large circuits and long proving times (typically orders of magnitude slower than DSL-based systems).

Domain-Specific Systems such as zkDBs [37, 58, 95] and zkML frameworks [18, 61, 80, 91] optimize for fixed workloads with known computational patterns. They achieve impressive efficiency by tailoring circuits and proof schemes to their domains but sacrifice generality. They lower each operator of a fixed vocabulary with efficient constructions. Their shapes, however, are fixed or padded to bounds that the user or a per-operator rule provides. These frameworks cannot express analytical logic beyond their predefined operations.

Circuit compilers for general programs, such as CirC [65], go further and prove loop bounds with an SMT solver. None of these systems automatically derives the capacities, index ranges and other facts of a program, or chooses constructions for its operators by such facts.

3 Research Overview

3.1 Problem Statement

Programs and input domains. A program P is a Python function that uses NumPy, together with a declared *input domain* D. The input domain gives the shape of each input array (or its maximum size), the type of each element, which determines its range, and optional preconditions on the inputs. An input u is valid when $u \in D$. We write $[\![P]\!](u)$ for the result of running P on u with NumPy, with reals in the fixed point of Sec. 7, and $[\![P]\!](u) = \bot$ when $u \notin D$ or the run fails, for example because an assertion does not hold.

The problem. Compiling P produces a circuit C_P, a set of constraints over the outputs y, which are public, the inputs u, which are private, and auxiliary variables w, which hold the intermediate values that the prover supplies, so the witness of Sec. 2.1 is (u, w). Our goal is to recover expressiveness while still compiling programs to fixed arithmetic circuits. We seek a circuit that is correct, has a fixed structure, and is as small as possible. The circuit is *correct* when, for every input,

$$\forall u.\, \forall y. \quad \big(\exists w.\, C_P(u, y, w)\big) \iff y = [\![P]\!](u).$$

Read from right to left, this equivalence states completeness: every correct output can be proved. Read from left to right, it states soundness: no choice of w makes a wrong output provable, and a failing run has no proof at all. The circuit has a fixed structure when C_P depends only on P and D: for all inputs, the circuits have identical topology, and the input determines only the values of the variables. Its size determines the proving cost. Since a correct, fixed C_P is proved by the underlying proof system, compilation reveals nothing beyond what that system exposes.

3.2 Design Overview

Real-world analytics mixes numeric operations with array-programming patterns (broadcasting, slicing, masking, reduction) whose shapes and axes often depend on data. In zk-SNARKs, however, the compiler must emit a fixed and finite circuit: neither control flow nor memory access patterns can be data-dependent in a way that changes circuit topology. Every data-dependent shape and loop count therefore needs a bound that holds on all of D, which is precisely what the compiler needs to pre-allocate a finite circuit and to bound all loops in the lowered code. Such a bound is one kind of fact: a claim φ about runtime values together with a condition g, such that on every valid input, φ holds whenever g does. Examples are "$0 \le k < 16$", "`len(rows)` ≤ 5" and "`p` is a permutation of $0, \ldots, n-1$". We obtain facts by symbolic execution and SMT solving (Sec. 4), and reject a program if some bound cannot be proved.

The challenge is not to invent new array operators, but to make common data-analytic operators compilable under the fixed-structure requirement. Array operators are difficult because they couple three concerns: (i) they allocate intermediate storage whose size depends on shape; (ii) they compute over data-dependent offsets and axes; and (iii) they often include data-dependent operations (e.g., slicing and filtering). A bound fixes the size of the storage, but a data-dependent access also needs a construction that is valid on all of D, and a small circuit needs a cheap one. Since validity depends on facts (Sec. 2.2), we keep each array operator whole in the intermediate representation (IR) until the facts about its operands are known. A whole operator also exposes a relation that the circuit can check instead of computing, such as the order of a sort. Only then do we lower the operator, choosing among known constructions by these facts and by their cost on the target backend (Sec. 5).

Relying on a fact, however, is sound only if the circuit enforces it. Since the prover chooses every value that the constraints leave free (Sec. 2.1), a fact that holds in every run of P may still fail in a proof unless every assignment that satisfies the constraints of C_P also satisfies it, that is, unless C_P *enforces* it. A range check on an input, an assertion, the constraints that define an operator or those of a selected construction may enforce a fact. Constructions thus both rely on facts and enforce them, so a construction chosen on its own may rely on a fact that nothing enforces, or pay for one that another construction already enforces. Two constructions may even rely on each other, so that neither fact is checked. We therefore choose all constructions together, as a single selection problem in which every fact relied on is enforced elsewhere in the circuit, without circular reasoning. Sec. 5 shows that the whole circuit is then correct whenever each construction is.

3.3 Scope

As a key design consideration, we aim to provide a language that is intuitive and easy to use for data analysts. To this end, we follow the design principles in Python where the programs written in our language are also valid Python programs. A program is a function whose parameters declare its input domain. It supports data-dependent programs with implicitly bounded control flow (e.g., breakable loops and multi-exit functions), and variables rebound to values of a different type. Developers annotate inputs with type hints, which determine circuit layout and data representation. To simplify the presentation, we treat the inputs as private and the result as public. The language supports common elementary data types, including integers, reals, booleans, lists, tuples, and bounded-dynamic multidimensional arrays. A complementary goal is to provide first-class array-programming primitives for common analytical patterns so developers need not rebuild them from scratch. Zinnia offers real-valued arithmetic, nonlinear functions, and multidimensional array operations over bounded-dynamic arrays, along with utilities for transformation, reduction, etc. Data-analytic primitives are invoked through familiar NumPy- or Python-style APIs: the compiler supports 168 NumPy functions together with Python's `math` module. Calls into opaque libraries

such as Pandas or PyTorch are outside the language, since the compiler must reason about every operation to bound it. We compile a program to a circuit directly instead of proving it in a zkVM, since a proof is generated for every new input and emulation systems must pay for general-purpose CPU logic at every instruction that the processor executes. Faster hardware, such as GPUs, speeds up circuit provers as well as zkVMs. A zkVM remains the better choice for programs that call opaque libraries, which our compiler cannot bound.

Any system compiling dynamic behavior to valid fixed circuits inherently requires pre-allocating resources for the worst-case execution path. A program whose worst case far exceeds its typical run therefore needs a circuit much larger than the run. Since a bound exists only if the program terminates on every valid input, whether one exists is undecidable [87], and every such compiler rejects some programs that do have a bound: compilation can be *sound* but not complete for all statically bounded array programs. We prove bounds by symbolic execution with budget-bounded SMT solving [22, 63], and reject a program when a bound cannot be proved, naming the quantity involved. A precondition that bounds the input often makes such a program acceptable.

Every supported operator has a construction that relies only on its source semantics, so lowering accepts every program that symbolic execution accepts. Sec. 8 measures how many programs of three corpora the compiler accepts, and why it rejects the others.

4 Fact Inference by Symbolic Execution

The compiler performs *symbolic execution* over the program's input domain: it proves that control flow is bounded, that array shapes stay within their capacities, and that program quantities that affect circuit structure are bounded at compile time. It then records the necessary data dependencies and lowers control flow to a branch-free, SSA-style intermediate representation (IR), in which array operators stay whole, and infers facts about the program's values along the way.

4.1 Facts on Scalars and Arrays

A fact is a pair (φ, g) as defined in Sec. 3.2. Facts about scalars are intervals (e.g., $0 \le i < n$) and relations between two values (e.g., $d \neq 0$, $x = y$, $r_0 \le r_1$). Since array programs compute on the elements of arrays, we also introduce facts about arrays, which state what every element of an array, or of a part of it, satisfies. The condition is usually the path condition of a statement: the conjunction of its enclosing branch conditions, the activity of its loop iteration, and the conditions that no `break` or `return` has occurred before it.

At program entry, the input domain bounds every input, and a precondition of the program may restrict the inputs further, for example, `forall(a, in_range(0, 9))`. The semantics of each operator gives facts about its result, such as $|x| \ge 0$, $\texttt{argmax}(x) \in [0, n)$, or the order of the elements that `sort` returns. A successful run, moreover, has passed every point at which the program could have failed, so the conditions tested there hold afterwards. For example, after a read $x[i]$, $-n \le i < n$ holds, since the read fails otherwise, and after `assert c`, c holds.

We represent array facts with the segmentations of FunArray [24]. A segmentation splits the positions of an array into consecutive segments with bounds $b_0, \ldots, b_k$, written $\{b_0\}\ E_1\ \{b_1\} \cdots E_k\ \{b_k\}$, and states for each segment the element facts E_j that every cell in it satisfies: a range, or an order between neighboring cells. An array fact describes one array value, so views and slices inherit the facts of the array they come from.

The semantics of an operator also gives array facts: `argsort` returns a permutation, `flatnonzero` increasing positions, and the inverse indices `inv` of `unique` lie in $[0, \texttt{count} - 1]$, where `count` is the runtime number of distinct values. Such facts prove preconditions about whole arrays. A one-hot matrix built as `b[arange(n), inv] = 1` has `count` columns, so the facts of `inv` prove every write in range, and the IR needs no assertion.

(a) Types and facts

$$\tau ::= s \mid [\tau_1, \ldots, \tau_n] \mid (\tau_1, \ldots, \tau_n) \mid \langle s;\, \vec{d} \rangle \mid \langle s;\, \vec{d} \mid L \rangle \qquad s ::= \texttt{Int} \mid \texttt{Real} \mid \texttt{Bool}$$
$$\varphi ::= b \le x \le b \mid x \ne 0 \mid x \le y \mid x = y \mid perm(a) \mid a \triangleright \mathcal{S} \qquad b ::= c \mid x + c$$
$$\mathcal{S} ::= \{b_0\}\ E_1\ \{b_1\} \cdots E_k\ \{b_k\} \qquad E ::= [b, b] \mid \le \mid < \mid E \wedge E$$
$$\sigma ::= \langle \Gamma, F, p, I \rangle \qquad F ::= \{(\varphi_1, g_1), \ldots, (\varphi_m, g_m)\} \qquad \iota ::= v = f(\vec{v}) \mid v = \texttt{select}(c, v, v) \mid \texttt{assert}_g\ \varphi$$
$$\vec{d} = d_1, \ldots, d_k \qquad \langle s;\, \vec{d} \rangle \sqcup \langle s';\, \vec{d} \rangle = \langle s \sqcup s';\, \vec{d} \rangle$$
$$\langle s;\, \vec{d} \mid L \rangle \sqcup \langle s';\, \vec{d}' \mid L' \rangle = \langle s \sqcup_a s';\, \vec{e} \mid \textstyle\prod_i e_i \rangle \ \text{ with } e_i = \max(d_i, d_i'),\ s \sqcup_a s' \in \{\texttt{Int}, \texttt{Real}\}$$

(b) Generic rules

$$\textsc{Req-Proved}\ \frac{F \vDash_p \varphi}{\sigma \vdash \text{require } \varphi \dashv \sigma} \qquad \textsc{Req-Assert}\ \frac{F \nvDash_p \varphi}{\sigma \vdash \text{require } \varphi \dashv \sigma \oplus_p \varphi \lhd \text{assert}_p\ \varphi}$$

$$\textsc{Op}\ \frac{\sigma \vdash \vec{e} \Downarrow \vec{v} : \vec{\tau} \dashv \sigma_1 \qquad spec(f) = \{Req\}\ f(\vec{\tau}) : T\ \{Ens\} \qquad \sigma_1 \vdash \text{require } Req(\vec{v}) \dashv \sigma_2 \qquad v \text{ fresh}}{\sigma \vdash f(\vec{e}) \Downarrow v : T_{F_2,p}(\vec{\tau}, \vec{v}) \dashv \sigma_2 \oplus_p Ens(\vec{v}, v) \lhd v = f(\vec{v})}$$

$$\textsc{If}\ \frac{\sigma \vdash e \Downarrow c \dashv \sigma_0 \quad \sigma_0[p \wedge c] \vdash s_1 \dashv \sigma_1 \quad \sigma_0[p \wedge \neg c] \vdash s_2 \dashv \sigma_2}{\sigma \vdash \texttt{if}\ e\texttt{:}\ s_1\ \texttt{else:}\ s_2 \dashv \langle \Gamma_1 \sqcup_c \Gamma_2,\ F_1 \cup F_2,\ p,\ I_1 \cup I_2 \rangle} \qquad \textsc{While-Step}\ \frac{\sigma \vdash e \Downarrow c \dashv \sigma_0 \qquad F_0 \nvDash_p \neg c \qquad \sigma_0[p \wedge c] \vdash s\texttt{;}\ \texttt{while}\ e\texttt{:}\ s \dashv \sigma_1}{\sigma \vdash \texttt{while}\ e\texttt{:}\ s \dashv \sigma_1[p]}$$

$$\textsc{For-Range}\ \frac{\sigma \vdash e_1, e_2 \Downarrow a, b \dashv \sigma_0 \qquad K = \max(0, \max_{F_0,p}(b - a)) \qquad \sigma_k[p \wedge a + k < b][x \mapsto a + k] \vdash s \dashv \sigma_{k+1} \qquad \sigma_K[p] \vdash \text{require } a + K \ge b \dashv \sigma'}{\sigma \vdash \texttt{for}\ x\ \texttt{in}\ \texttt{range}(e_1, e_2)\texttt{:}\ s \dashv \sigma'} \qquad \textsc{While-Exit}\ \frac{\sigma \vdash e \Downarrow c \dashv \sigma_0 \qquad F_0 \vDash_p \neg c}{\sigma \vdash \texttt{while}\ e\texttt{:}\ s \dashv \sigma_0}$$

(c) Specifications of selected operators

$$\{\top\}\ \texttt{flatnonzero}(h : \langle \texttt{Bool};\, n \rangle) : \langle \texttt{Int};\, n \mid n \rangle\ \ \{\ell_r = \textstyle\sum_i h_i \wedge r \triangleright \{0\}\ [0, n-1] \wedge < \{\ell_r\}\ [0, 0]\ \{n\}\}$$
$$\{\top\}\ \texttt{arange}(m : \texttt{Int}) : \langle \texttt{Int};\, L \mid L \rangle\ \ \{\ell_r = \max(0, m) \wedge r \triangleright \{0\}\ [0, L-1] \wedge < \{L\}\}$$
$$\text{with } L = \max(0, \max_{F,p}(m))$$
$$\{\top\}\ x[h : \langle \texttt{Bool};\, n \rangle] : \langle s;\, n \mid n \rangle\ \ \{\ell_r = \textstyle\sum_i h_i \wedge r \triangleright \{0\}\ rng(x) \sqcup [0, 0]\ \{n\}\} \text{ with } x : \langle s;\, n \rangle$$
$$\{\top\}\ \texttt{sort}(x : \langle s;\, n \rangle) : \langle s;\, n \rangle\ \ \{r \triangleright \{0\}\ rng(x) \wedge \le \{n\}\}$$
$$\{\top\}\ \texttt{argsort}(x : \langle s;\, n \rangle) : \langle \texttt{Int};\, n \rangle\ \ \{perm(r)\}$$
$$\{\top\}\ \texttt{amax}(x : \langle s;\, n \rangle) : s\ \ \{r \in rng(x)\},\ \text{likewise } \texttt{amin}$$
$$\{\top\}\ \texttt{concatenate}((x, y)) : \langle s;\, n + n' \mid L + L' \rangle\ \ \{\ell_r = \ell_x + \ell_y\} \text{ with } x : \langle s;\, n \mid L \rangle,\ y : \langle s;\, n' \mid L' \rangle$$
$$\{-n \le i < n\}\ \texttt{read}(x : \langle s;\, n \rangle,\ i : \texttt{Int}) : s\ \ \{r \in rng(x)\}$$
$$\{-n \le i < n\}\ \texttt{write}(x : \langle s;\, n \rangle,\ i : \texttt{Int},\ v : s) : \langle s;\, n \rangle\ \ \{r \triangleright \mathcal{S}_x \sqcup rng(v)\}$$

Fig. 4. **Symbolic execution.** (a) Types and facts. (b) Generic rules. (c) Specifications of selected operators.

Fact inference is incomplete, but a fact is used only where it holds. A missing fact may leave a bound unproved or make the circuit larger, but never incorrect.

4.2 Symbolic Execution

To infer facts, we must know, at every point of the program, which values each variable may hold and under which condition that point is reached. We therefore use symbolic execution to compute with symbolic values while accumulating path conditions, so we can turn control dependence into guarded data dependence and extract semantic bounds. Symbolic execution forks at every branch, so naive exploration enumerates up to 2^b paths over b conditional branches, the well-known *path explosion*. We avoid explicit path enumeration by evaluating both sides of a branch and joining the alternatives into a single symbolic state with selects on the branch condition at control-flow meet

points [55]. This design eliminates exponential path enumeration while preserving path-dependent information within a single state. Empirically, this strategy scales to realistic analytical workloads (Sec. 8.1). However, expressions may still grow due to deeply nested conditionals, thus requiring bounded solving as discussed below. Moreover, the symbolic values are IR expressions, so the same evaluation also produces the program's IR. When array operations and control flow are intertwined, the compiler must keep data dependencies precise while giving the IR a fixed structure.

We maintain a symbolic state σ during compilation to model the program's execution [50] with symbolic values, facts and path conditions, and summarize it through two judgments:

$$\sigma \vdash e \Downarrow v : \tau \dashv \sigma' \qquad \sigma \vdash s \dashv \sigma' \qquad \sigma = \langle \Gamma, F, p, I \rangle.$$

Expression evaluation produces an IR value v of type τ and an extended state, and statement evaluation transforms the symbolic state itself. Conceptually, this state has four components: the symbolic environment Γ, which maps program variables to IR values, the set of facts F, the path condition p, and the IR I produced so far, as SSA-style statements. We write $F \vDash_p \varphi$ when the facts whose conditions p implies prove φ, $\sigma[p']$ and $\sigma[x \mapsto v]$ for σ with path condition p' or with x bound to v, $\sigma \oplus_g \Phi$ and $\sigma \oplus \Phi$ for σ with the facts Φ added under condition g or unconditionally, and $\sigma \lhd \iota$ for σ with the IR statement ι added. In Figure 4, rule names are violet, and a rule adds facts in teal and IR statements in blue.

Figure 4(a) defines the objects that the state holds. Zinnia is statically typed, as in existing zk-SNARK DSLs [7, 21, 64]. Static typing is essential because the compiler must construct a fixed arithmetic circuit: the circuit structure (i.e., constraint layout) cannot depend on runtime values. Users must explicitly declare types for inputs, while the compiler infers types for subsequent expressions without requiring annotations, following conventional local type inference [72]. Our goal is not to propose a new type system. We therefore omit standard rules and focus on the points needed to support bounds and joins. Scalar types include `Int`, `Real`, and `Bool`. Composite types are recursive and include lists and tuples. Arrays range only over scalar element types s, and composite elements are expressed via lists/tuples. An array type is either static or bounded-dynamic, and the join $\sqcup$ merges two array types at the end of a branch. An IR statement defines a value by an operator or a select, or asserts a claim under a path condition.

The generic rules in Figure 4(b) translate operators, branches and loops. Each supported operator has a specification $\{Req\}\ f(\vec{\tau}) : T\ \{Ens\}$, whose precondition states when the operator succeeds and whose postcondition states what its semantics guarantees about the result. The rule Op evaluates the operands, emits a statement that defines the result, gives it the type T, which may depend on the facts, and adds Ens to the facts. The rules Req-Proved and Req-Assert handle the precondition and the program's own `assert` statements. If the facts prove the condition, the IR needs nothing. Otherwise, the IR asserts it under the current path condition, so that the translated program fails exactly where the source program does. Either way, the condition becomes a fact for later code.

Control Flow. Conditionals compile by first lowering the guard to a symbolic boolean c, then symbolically executing each branch under $p \wedge c$ and $p \wedge \neg c$, and finally joining the two states (If): $\Gamma_1 \sqcup_c \Gamma_2$ merges each variable by a select on c, its type by $\sqcup$, and its segmentations by their join, and the facts of both sides are kept under their conditions. This yields a single state in which control dependence is represented as data dependence. Writes are guarded with the path condition, so off-path updates become no-ops. This is the same mechanism used for ordinary assignments, lifted to array cells. Loops compile by inductive unrolling: the While-Step rule executes one body iteration and recurs, while While-Exit terminates when the facts prove the guard false. Calls to the program's own functions are inlined, and multiple return sites yield a single symbolic result guarded by the corresponding path conditions. This structure supports

early returns: a `return` both contributes a guarded candidate to the result and refines the active path condition to disable subsequent statements on that path. `break` and `continue` likewise refine the path condition of the rest of the loop. Recursion is supported when it is statically bounded: the compiler discharges a boundedness obligation using SMT-backed reasoning, ensuring that recursive unfolding terminates within a configurable unrolling limit, and stops once the facts prove a call unreachable. If boundedness cannot be established, the program is conservatively rejected to preserve soundness, consistent with prior ZKP compilers [65].

Finally, Figure 4(c) gives the specifications of selected operators, in which r denotes the result, ℓ_r its runtime length, and $rng(x)$ the hull of the element ranges of x. The result of `flatnonzero` or of a filter $x[h]$ receives a capacity from the type of its operand, a runtime length, and a segmentation of its cells. Array creation such as `arange` infers a bounded-dynamic type when some extents are unknown but the facts yield a finite upper bound on total elements, which becomes its capacity, so the type depends on facts. A read $x[i]$ requires its index to be in range and yields a value in the range of x, which is how an array fact becomes a fact about a scalar, as `amax` does. Since both sides of a branch are evaluated, a read may lie on a path that the run does not take, where its index may be out of range. Unless the facts prove the index in range on every path, the IR therefore reads at the masked index $\texttt{select}(p, i, 0)$, after wrapping a negative i to $i + n$. A write yields a new array whose segmentation $\mathcal{S}_x \sqcup rng(v)$ joins the range of the written value v into every segment of x, so no fact about the old cells survives unchanged.

The rules ask two kinds of questions of the facts: whether they prove a claim ($F \vDash_p \varphi$), and the largest value of a term that they admit ($\max_{F,p}$). We answer each by constant folding, interval propagation over the IR and the scalar facts, the array facts of the arrays that a value is read from, and finally an SMT query to Z3 [26], trying each only when the cheaper ones fail. Array facts never reach the solver as quantified formulas, because a segmentation already states what holds for every cell, and a query includes only the facts that share variables with the question. A question that cannot be answered within the solver's step budget yields unknown, which is never read as false: the assertion stays in the IR, or the compiler conservatively rejects the program.

4.3 Bounds and Capacities

zk-SNARKs mandate a finite, fixed circuit, so array bounds, loop bounds, and other structure-critical quantities must be determined at compile time. Purely syntactic restrictions guarantee this, but they reject many idiomatic analytics programs whose bounds are implicit in the dataflow. We derive from the facts a bound on the size of every array whose shape depends on the input, and on the trip count of every loop.

Loops. A `for` loop over a range with constant bounds is unrolled exactly. When its start or stop depends on the input, we unroll it K times, the largest number of iterations that the facts admit (For-Range), and iteration k runs under the path condition that its index lies before the stop. A loop over an array with a runtime length runs once per entry of its first axis, each iteration under the condition that the cell is active. A `while` loop whose guard is not proved false within the unrolling limit is rejected.

Capacities. As Figure 4(a) shows, Zinnia supports both statically shaped arrays $\langle s;\ d_1, \ldots, d_k \rangle$, whose shape is known at compile time, and bounded-dynamic arrays. Static arrays enable precise compilation when shape-changing primitives (e.g., stacking or reduction) can be resolved through compile-time shape reasoning. When a shape- or axis-critical quantity cannot be resolved, we conservatively fall back to a bounded-dynamic type.

A *bounded-dynamic* array x of type $\langle s;\ d_1, \ldots, d_k \mid L \rangle$ has a shape that depends on the input. Its rank k is known, but its extent $|x|_i$ along each axis i is a runtime value of at most d_i, and its length

$\ell_x = \prod_i |x|_i$ is at most its *capacity* $L \le \prod_i d_i$, the number of cells that the circuit reserves for it. The bounded-dynamic form is intentionally restricted: it fixes the element type and the rank while recording compile-time bounds on each axis and on the total element count. These bounds are exactly what the compiler needs to pre-allocate a fixed circuit that can accommodate any runtime shape within the bounds, while still rejecting programs whose potential shapes would require an unbounded circuit. The bounds of the axes keep an array of capacity 30 within a 5×6 grid, which rules out a 30×1 array. In the example of Figure 3(a), for instance, `im` has the static type $\langle$`Int`; $5, 6\rangle$, `rows` has the bounded-dynamic type $\langle$`Int`; $5 \mid 5\rangle$, and the cropped image has the bounded-dynamic type $\langle$`Int`; $5, 6 \mid 30\rangle$.

Each operator that produces a bounded-dynamic array derives its bounds from the types of its operands and from the facts. In Figure 4(c), the result of `flatnonzero` has one cell per flag, and its length is the number of set flags. The capacity of `arange(m)` is the largest value of m that the facts admit, and a slice with runtime bounds keeps the bounds of the sliced array. Appendix A.3 gives the bounds of transposes, reshapes, concatenations and reductions. If the facts admit no largest value for a bound, the program is rejected with the name of that quantity, as for a loop. Cells beyond the runtime shape hold padding, and every operator that reads such an array ignores them: a reduction, for instance, replaces each inactive cell by the identity of its operation.

Joins. Symbolic execution with merged states yields a valid circuit only if the merged values have a single, well-defined representation. At a branch join, a variable may have multiple reaching assignments, and the compiler must pick a single type for the guarded merge that follows. In the example of Figure 3, `result` is an empty array on one side and the cropped image of type $\langle$`Int`; $5, 6 \mid 30\rangle$ on the other. We model this with a partial join operator $\sqcup$, defined only on compatible types: at merge points, the merged type must be defined as $t_1 \sqcup t_2$, otherwise the program is rejected as type inconsistent where the variable is used. Our join is defined within type families and is deliberately conservative across families. Scalars join by standard numeric promotion (e.g., `Bool` $\sqcup$ `Int` = `Int` and `Int` $\sqcup$ `Real` = `Real`). Arrays join by joining element types and taking the max of each axis bound: two static arrays of the same shape keep their type, while two 1-D arrays of different lengths, or a bounded-dynamic array and one of the same rank, yield a bounded-dynamic type with the product of these bounds as its capacity and a runtime shape selected by the branch condition. An array that is empty at compile time, such as `np.array([])`, joins as a bounded-dynamic array with zero bounds, so `result` has the type $\langle$`Int`; $5, 6 \mid 30\rangle$. Lists and tuples of numbers join as bounded-dynamic arrays. Other joins are rejected where the variable is used, among them arrays of different ranks and static arrays of different shapes with several axes, preventing ill-typed merges that would require union types or dynamic tagging. Appendix A gives the full rules.

5 Choosing Constructions for Array Operators

Symbolic execution leaves every array operator of the IR whole. This section lowers the IR to constraints. An array operator has several constructions, which differ in cost and in the facts they rely on and enforce, so we select the constructions of all operators together and prove each construction correct once.

5.1 Array Operators

An array operator takes arrays as operands and computes an array or a scalar from them, such as reads and writes at runtime indices, slices with runtime bounds, sorts, searches and matrix products. An occurrence of an operator in the IR is an *instance*, which must be expressed by constraints over the cells of its operands, in one of many ways. Figure 5(a) shows three ways to read $x[i]$ at a

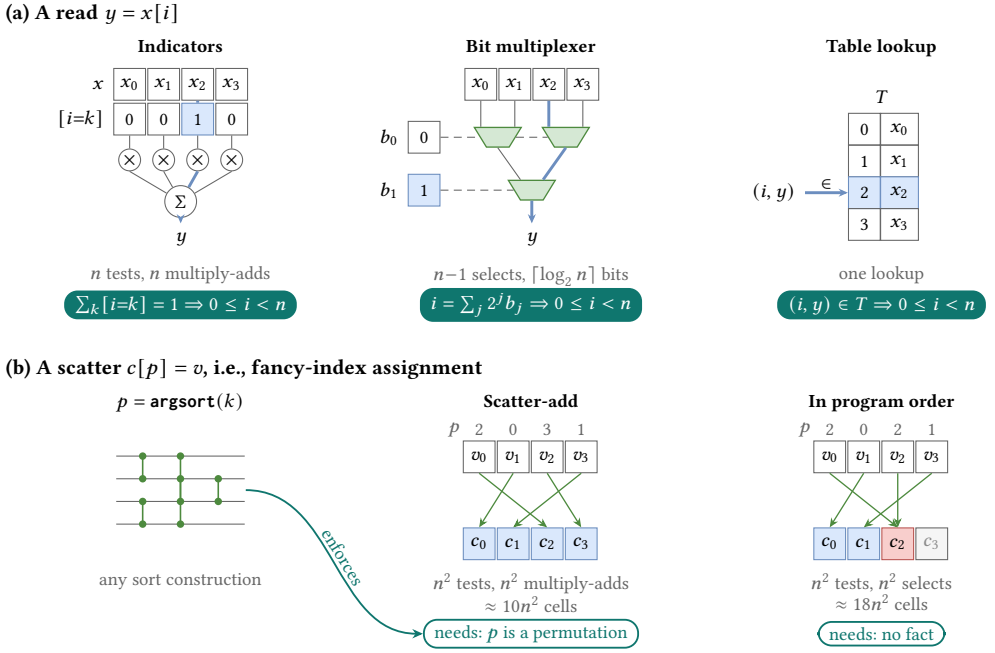


Fig. 5. **Constructions of two array operators, for** $n = 4$. (a) A read at $i = 2$. (b) A scatter, with costs in circuit cells. Filled pills are facts that a construction enforces, outlined pills facts that it relies on.

runtime index i from an array x of n elements. They differ in how they locate the element, and this decides their cost. Indicators compare i with every position k, which costs n equality tests, and the element is the sum of the cells weighted by the outcomes. A multiplexer instead decomposes i into $\lceil \log_2 n \rceil$ bits, each of which halves the remaining cells, so it needs $n - 1$ selects and no equality test. A table lookup leaves the search to the proof system, which checks that the pair $(i, x[i])$ is a row of a table holding the cells of x. The cheapest depends on n, the number of reads and the target.

A fact about the operands can make a simpler construction valid. Figure 5(b) shows a scatter $c[p] = v$, which writes each v_k to the cell c_{p_k}. Without a fact about p, two writes may target the same cell, as v_0 and v_2 do on the right, and the later one must win. The construction must then apply the writes in program order, with a select per cell and write. If p is known to be a permutation, no two writes collide, so the scatter equals a scatter-add, which computes each cell as the sum $c_j = \sum_k [p_k = j]\, v_k$ of the values written to it, in any order (left).

Constructions can also end up *relying on each other* (Sec. 3.2). A sort that takes the order p from the prover and checks that the keys $x[p_0], x[p_1], \ldots$ increase reads x at the positions p_r. A cheap read assumes that p_r is in range, because p is a permutation, but the sort shows that p is a permutation only if these reads are correct. A dishonest prover could then choose an out-of-range p_r, so the sort checks $0 \le p_r < n$ itself, and the selection rules out such cycles in general.

Even when every choice is correct, choosing a construction for each operator on its own makes the circuit larger than necessary, because *the cheapest choice for one operator often depends on the choice for another*. In Figure 5(b), the scatter-add computes the tests $[p_k = j]$ for every k and j. When the written values are $0, \ldots, n - 1$, these tests are exactly the one-hot vector of each c_j, which a later read at index c_j can use instead of comparing c_j with every position again. Some constructions also *share a cost that is paid once per circuit*: a check with a random value from the

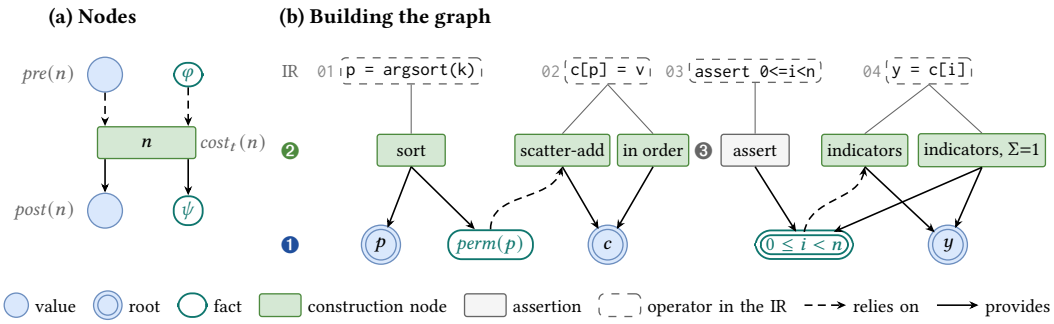


Fig. 6. **The graph of requirements and construction nodes.** (a) The two kinds of node. (b) The graph built for the IR of a small program, before any node is selected.

verifier needs a second phase of the circuit, and a memory argument needs one trace of all memory accesses, so they pay off only once several operators share them. Likewise, a read that checks its own index makes the program's range assertion unnecessary, and the circuit should not pay for both. We therefore select the constructions of all operators together.

5.2 Requirements and Constructions

To select constructions together, we need a common representation of what each construction relies on, what it provides and what it costs. We use a bipartite graph of *requirements*, the things that the circuit must contain, and *construction nodes*, the ways of providing them.

A requirement is a value that the circuit must contain or a fact that it must enforce. Stating requirements explicitly lets one construction supply what another needs, and lets the selection pay for each requirement once. They include the output elements that the program uses, the facts that constructions rely on, and values that several constructions use, such as the bits of an index read by several multiplexers.

A construction node n is one way of providing requirements. It relies on some requirements, its *preconditions* $pre(n)$, and provides others, its *postconditions* $post(n)$. For example, the scatter-add has the fact that p is a permutation as its precondition and the cells of c as its postconditions. Each construction also has a cost model, which estimates the cost $cost_t(n)$ of each of its nodes on each target t, and an emitter, which generates the constraints of a node once the node is selected. A node n *provides* a requirement r if r is one of its postconditions or, for a fact (φ, g), if one of its postconditions is a fact (φ', g') such that g implies g' and φ' implies φ. Implication matters because constructions state facts about the same value in different forms and under different conditions.

The graph is built from the IR once symbolic execution has finished, as Figure 6(b) shows for a small program. The output elements that the program uses become requirements ❶, the roots of the graph. Every construction offers nodes for each instance that it applies to ❷, whose preconditions and postconditions add further requirements, such as the fact that p is a permutation. Each assertion of the IR becomes a node ❸, whose postcondition is its claim and whose cost is that of its constraints, and its claim becomes a root. The read whose indicators are checked to sum to one provides this claim as well, while the read without that check relies on it, so selecting the checked read makes the assertion unnecessary. A precondition that no node provides receives a node of its own, which asserts the fact under its path condition. Every other IR statement, such as scalar arithmetic or a select, becomes one node for its standard encoding, whose postconditions are its value and the facts that its constraints enforce on their own, and which every selection contains.

(a) Constructions that compute their result

Indicators	$\{\top\}\ y = x[i] \rightsquigarrow$	$e_k = [i = k],\ \ y = \sum_k e_k x_k,\ \ \sum_k e_k = 1$ $\{y = x[i] \wedge 0 \le i < n\}$
Running-Products	$\{h \in \{0,1\}^n\}\ r = first(h) \rightsquigarrow$	$L_k = L_{k-1}(1 - h_k),\ \ L_{-1} = 1,\ \ r = \sum_k L_k,$ $e_k = L_{k-1} - L_k$ $\{r = first(h) \wedge e = E(r)\},\ \ \bigvee_k h_k$ asserted
Shift	$\{e = E(r)\}\ z = x[r : r + w] \rightsquigarrow$	$z_u = \sum_d e_d\, x_{u+d}\quad \{z = x[r : r + w]\}$
Scatter-Add	$\{perm(p)\}\ c[p] = v \rightsquigarrow$	$c_j = \sum_k [p_k = j]\, v_k$ $\{c = scatter(p, v)\},\ \ \{([p_k = j])_k = E(c_j)\}$ if $v_k = k$
One-Hot	$\{e = E(i)\}\ y = x[i] \rightsquigarrow$	$y = \sum_k e_k x_k\quad \{y = x[i]\}$
Perm-Hits	$\{\top\}$ assert $perm(p) \rightsquigarrow$	$\sum_k [p_k = j] = 1$ for every $j < n\quad \{perm(p)\}$
Masked-Write	$\{\top\}\ a' = a[m \mapsto v] \rightsquigarrow$	$a'_j = \texttt{select}(m_j, v_j, a_j)$ or, for a chain on ≤ 3 atoms, $a'_j = \sum_{C:\lambda_C \neq 0} [C]\, \lambda_C$ over cases $C\ \ \{a' = a[m \mapsto v]\}$
Arg-Extreme	$\{\top\}\ u = argmax(x) \rightsquigarrow$	$c_j = [x_j > b_{j-1}],\ \ b_j = \texttt{select}(c_j, x_j, b_{j-1}),\ \ b_0 = x_0$ $u_j = \texttt{select}(c_j, j, u_{j-1}),\ u_0 = 0\ \ \{u_{n-1} = argmax(x)\}$
Arg-Extreme-Sorted	$\{x \triangleright \{0\} \le \{n\}\}\ u = argmax(x) \rightsquigarrow$	$u = \sum_{j<n-1} [x_j < x_{n-1}]\quad \{u = argmax(x)\}$

(b) Constructions that check values supplied by the prover

Sort-Hint	$\{\top\}\ p = argsort(x) \rightsquigarrow$	$p = \hat{p},\ \ 0 \le p_r < n,\ \ (x[p_r], p_r) < (x[p_{r+1}], p_{r+1})$ $\{p = argsort(x) \wedge perm(p)\}$
Search-Hint	$\{x$ non-decreasing$\}\ u = search(x, t) \rightsquigarrow$	$u = \hat{u},\ \ 0 \le u \le n,\ \ u > 0 \Rightarrow x_{u-1} \le t,$ $u < n \Rightarrow t < x_u\ \ \{u = search(x, t)\}$

(c) Constructions that check with a verifier challenge γ

Sort-Product	$\{\gamma\}\ s = sort(K) \rightsquigarrow$	$s = \hat{s},\ \ s_r < s_{r+1},\ \ \prod_j (\gamma - K_j) = \prod_r (\gamma - s_r)$ $\{s = sort(K)\},\ \ K_j = x_j 2^b + j,\ \ b = \lceil \log_2 n \rceil$
Perm-Product	$\{\gamma\}$ assert $perm(p) \rightsquigarrow$	$\prod_k (\gamma - p_k) = \prod_k (\gamma - k)\quad \{perm(p)\}$
Freivalds	$\{\gamma\}\ M = AB \rightsquigarrow$	$R = \hat{R},\ \ A(B\beta) = R\beta,\ \ M = \texttt{rescale}(R),$ $\beta = (1, \gamma, \gamma^2, \ldots)\quad \{M = AB\},\ \ A, B$ fixed-point codes

(d) Constructions for arrays read and written at runtime

Bit-Mux	$\{\top\}\ y = x[i] \rightsquigarrow$	$i = \sum_l 2^l b_l,\ \ b_l(1 - b_l) = 0,\ \ i < n,\ \ D = \lceil \log_2 n \rceil$ $m^0 = x,\ \ m^{l+1}_k = b_l\, m^l_{2k+1} + (1 - b_l)\, m^l_{2k},\ \ y = m^D_0$ $\{y = x[i] \wedge 0 \le i < n\}$
Table	$\{\top\}\ y = x[i] \rightsquigarrow$	$y = \hat{y},\ \ (i, y) \in \{(k, x_k) \mid k < n\}$ $\{y = x[i] \wedge 0 \le i < n\}$
Select-Write	$\{0 \le i < n\}\ c' = c[i \mapsto v] \rightsquigarrow$	$c'_j = [i = j]\, v + (1 - [i = j])\, c_j$ for every $j < n$ $\{c' = c[i \mapsto v]\}$
Insert	$\{\top\}\ a' = insert(a, x, e) \rightsquigarrow$	$h = \hat{h},\ \ h_k(1 - h_k) = 0,\ \ \sum_k h_k = 1,\ \ P = \sum_k k\, h_k,$ $(x - P)(x - P + n(1 - h_n)) = 0,\ \ g_k = \sum_{q<k} h_q,$ $a'_k = (1 - g_{k+1})\, a_k + g_k\, a_{k-1} + h_k\, e$ $\{a' = insert(a, x, e) \wedge -n \le x \le n\}$
RAM-Trace	$\{\tau \wedge 0 \le i < n\}\ \overline{a} \rightsquigarrow$	$S = \hat{S},\ \ S \equiv \overline{a},\ \ S$ sorted by (i, t), $i_r = i_{r-1} \wedge \text{read}_r \Rightarrow v_r = v_{r-1}$ $\{\overline{a}\},\ \ \overline{a} = \{(t, kind, i, v)\}$; includes initial writes

Fig. 7. **Specifications of selected constructions,** which (a) compute their result, (b) check values supplied by the prover, (c) check with a verifier challenge, or (d) read and write arrays at runtime. Facts are in teal.

5.3 The Construction Library

The library has 45 constructions in seven families, which cover sorting and searching, scans, windows and scatters, insertions and masked writes, element reads and arrays written at runtime, linear algebra, statistics, and permutation checks. Appendix A.4 lists them all. Figure 7 gives the specifications of selected constructions, in the form of the operator specifications of Figure 4(c). A

specification $\{Pre\}\ f \rightsquigarrow C\ \{Post\}$ states that the constraints C implement the operator f wherever the precondition holds and establish the postcondition. Hatted values come from the prover.

The constructions of Figure 7(a) take no value from the prover. Running-Products finds the first set flag r of a vector h from the products L_k, which equal 1 exactly before that flag, and their differences form the one-hot vector $E(r)$, from which Shift and One-Hot read without comparing r with every position. Such constructions also establish facts: Indicators establishes $0 \le i < n$ by checking that its indicators sum to one, and Perm-Hits that p is a permutation, which Scatter-Add relies on. Arg-Extreme-Sorted counts the elements below the last one of an array that a sort already ordered, so the cheapest argmax depends on another operator.

Checking a result is often cheaper than computing it, so the constructions of Figure 7(b) take the result from the prover and constrain it. Sort-Hint takes the stable order $\hat{p}$ of an argsort and checks with $n-1$ comparisons that the pairs $(x[p_r], p_r)$ increase, where a sorting network needs $O(n \log^2 n)$ comparisons [6]. Since n distinct pairs drawn from n elements are all of them, p is a permutation, and its own check $0 \le p_r < n$ avoids the cycle of Sec. 5.1. The checks of Figure 7(c) need a verifier challenge γ, drawn after the prover has committed to the values being checked, in a second phase of the circuit. Sort-Product and Perm-Product check rearrangements with a grand product, and Freivalds checks a matrix product with a random vector [32]. The challenge is a single requirement of the whole circuit, whose node accounts for the second phase at an estimated 738 cells.[1] Figure 7(d) covers arrays read or written at runtime positions. Bit-Mux computes the bits of an index once for all its reads, Table looks (i, y) up among the cells, and a written array either selects the new value in each cell at every write (Select-Write) or lives in a memory checked by a single trace τ of all accesses (RAM-Trace) [14], paid for once per circuit like the challenge.

The cost model of each construction is a formula over the shape of the instance, built from unit costs that we measured by compiling each construction on its own and counting its cells or gates.

5.4 Selecting Constructions and Correctness

A selection is a set S of construction nodes, and the circuit consists of their constraints. It must provide every root of the graph, computing every output element that the program uses and enforcing the claim of every assertion, so that an input on which the program fails has no proof. Each node of S is correct only when its preconditions hold, and in a proof a fact holds only if a constraint enforces it, even a fact that symbolic execution proves for every run (Sec. 3.2). Each precondition must therefore be provided by another node of S.

Even then, the providers may form a cycle. If one node relies on a fact φ to establish ψ, and another relies on ψ to establish φ, as the sort and its reads in Sec. 5.1 would, an assignment can violate both facts and still satisfy the constraints of both nodes. We therefore order the selected nodes by a rank $\rho(n) \in \mathbb{N}$ and require every precondition to be provided by a node of lower rank, so that it is enforced before the node that relies on it. If Figure 6(b) selects the sort, the scatter-add and the checked read, for instance, the scatter-add relies on the sort for the fact that p is a permutation, so it has rank 1, and the other two rank 0. Among these selections, we seek one of least estimated cost on the target t, counting each node once however many requirements it provides:

$$\min_{S,\rho}\ \textstyle\sum_{n \in S} cost_t(n)$$
$$\text{subject to } \forall r \in Roots.\ \exists n \in S.\ n \text{ provides } r,$$
$$\forall n \in S.\ \forall r \in pre(n).\ \exists m \in S.\ m \text{ provides } r \ \wedge\ \rho(m) < \rho(n).$$

[1] A construction that uses a challenge is sound except with probability at most $d/|\mathbb{F}_p|$, where d is the degree of its check (Schwartz–Zippel [75]), which is below 2^{-240} for every circuit we compile. It holds for the interactive protocol, which the proof system makes non-interactive with the Fiat–Shamir transform.

This problem resembles instruction selection [28, 42, 51] and the extraction of a cheapest program from an e-graph [94, 98], but there every requirement is a value, and a fact proved by an analysis can simply be assumed.

Every selection that meets these conditions gives a correct circuit, because each construction is proved correct once, against its specification, and the ranks let these proofs compose. A requirement *holds* under an assignment if it is a fact that is true wherever its path condition holds, or a value that equals what its specification states, such as an output element that equals what its operator computes. The verifier challenge and the RAM trace hold once they are provided. A node n is *sound* if every assignment that satisfies its constraints, and under which $pre(n)$ holds, makes $post(n)$ hold. For a node that uses the verifier challenge, soundness is probabilistic: for values fixed before the challenge is drawn, the probability that its constraints are satisfiable while $post(n)$ fails is at most $d/|\mathbb{F}_p|$. The node is *complete* if the prover can compute values that satisfy its constraints whenever its operands take their values in a successful run and $pre(n)$ holds. A construction is *correct* if its nodes are sound and complete.

Theorem 1. *Let C be the IR read as a circuit, in which every output element that the program uses equals the value that its operator computes, and let C' be the circuit obtained from C with a selection S that meets the conditions above. Suppose that C is correct and is satisfied by the values of every run that does not fail, that every construction in S is correct, and that every fact in the graph holds in every run that does not fail. Then C' is complete. If no node of S uses the verifier challenge, C' is also sound, and hence correct. Otherwise, for every input $u \in D$, every output $y \neq [\![P]\!](u)$ and all values fixed before the challenge, the probability over the challenge that C' is satisfiable is at most the sum of $d/|\mathbb{F}_p|$ over the checks of S.*

Appendix B proves the theorem. For soundness, the preconditions of the selected nodes hold in increasing order of rank, so every root is provided as specified. For completeness, the prover computes the values of the nodes in the same order. It also covers lowering in rounds. A construction can introduce array operators of its own, such as the reads $x[p_r]$ of Sort-Hint, and a later round selects constructions for them, with the circuit of one round as the IR of the next.

The theorem holds for every selection that meets the conditions, so the cost of a selection affects the size of the circuit but never its correctness. This matters because finding a selection of least cost is NP-hard: even without preconditions, it is weighted set cover [47]. The compiler therefore finds a cheapest selection when it can afford to. Whether a node is selected is a Boolean variable and its rank an integer variable, so the problem is in a form that Z3's optimizer [13, 26] accepts, and we limit its search to a fixed number of steps. For a large graph, or when the solver exceeds this budget, the compiler falls back to a greedy heuristic. Starting from the roots, the heuristic provides each requirement not yet provided with the cheapest node, counting the cheapest providers of its unprovided preconditions, and continues depth first with these.

6 Illustrative Walkthrough

We follow a variant of the example of Figure 3 through the compiler. In the program of Figure 8(a), the columns of the image a are stored sorted by `key`, and the program restores their original order before it crops. Lines 2–3 compute the rank c_j of each column, and line 4 reads the columns back in their original order. Lines 5–8 crop as in Figure 3, except that they index the first and last entries of `rows` and `cols` instead of taking their minimum and maximum, and that no branch handles an image without nonzero pixels. NumPy rejects such an image, since `rows[0]` is then out of range.

Symbolic execution. Since p is the result of `np.argsort`, it is a permutation ❶, and the assignment on line 3 becomes one scatter operator. Every c_j is one of the values of `np.arange(6)` ❷, so the

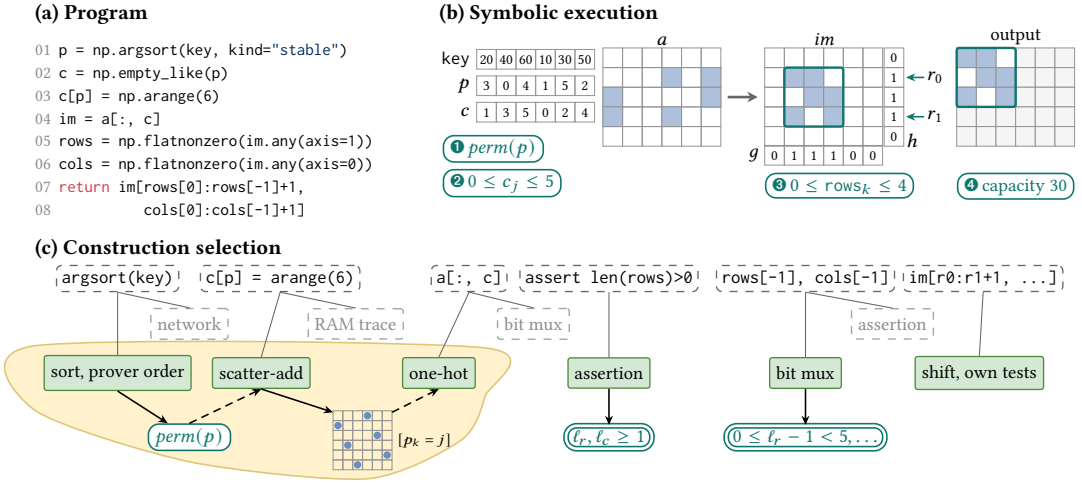


Fig. 8. **Compiling a variant of the example of Figure 3.** (a) The program. (b) Symbolic execution on one input. (c) Construction selection. Filled nodes are selected, dashed ones are not, and selected nodes that rely on each other are shaded.

reads on line 4 need no index check. The flags h and g mark the nonzero rows and columns of *im*. The IR builds `rows` from h with selects, as the positions of the set flags followed by padding, so its length is $\ell_r = \sum_i h_i$ and its entries lie in $[0, 4]$ ❸. Unlike `rows.min()` in Figure 3, which the compiler replaces by a scan for the first set flag, `rows[0]` reads the first cell r_0 and keeps its index check $\ell_r \geq 1$ as an assertion. The last set flag r_1 is read from `rows` at the runtime index $\ell_r - 1$, and the columns are handled alike. By ❸, the slice needs neither Python's wrap nor its clamp. The facts do not show $r_0 \leq r_1$, however, so the height of the output is computed as $\max(0, r_1 + 1 - r_0)$.

The slice becomes a window operator. Its output keeps the bounds of the sliced array and has the type $\langle \texttt{Int};\ 5, 6 \mid 30 \rangle$ ❹, so the cropped image fills the first $r_1 + 1 - r_0$ rows and $c_1 + 1 - c_0$ columns of 5×6 cells, and the remaining cells hold padding.

Selection. In the graph of Figure 8(c), the sort takes the order of the keys from the prover and checks it, which establishes that p is a permutation. The scatter-add relies on this fact, and its tests $[p_k = j]$ form the one-hot vector of each c_j, column j of the matrix in the graph. Each read on line 4 is then a sum $im_{ij} = \sum_k [p_k = j]\, a_{ik}$, and c is never computed. The reads at $\ell_r - 1$ and $\ell_c - 1$ use multiplexers, whose index bits prove these indices in range, so their assertions are not selected. The assertions $\ell_r \geq 1$ and $\ell_c \geq 1$ remain, and the window compares its bounds with every position.

Selected one operator at a time, the reads of each row of a count the scatter-add's tests as their own cost, which makes a multiplexer over the bits of c_j cheaper. The circuit then computes c and decomposes each c_j into bits, and nothing else changes. The circuit has 5,042 cells, against 5,594 with per-operator selection, and the difference lies in the reads on line 4.

7 Implementation

Zinnia is implemented as a compiler that lowers high-level data-analytic programs into arithmetic circuits. It consists of a Python front end, which parses the program and its input annotations, and a compiler core in Rust (about 77K lines of code), with a modular architecture that separates symbolic execution, IR optimization, lowering, and backend-specific synthesis.

Intermediate Representation. Zinnia compiles symbolic execution results into an SSA-style, branch-free linear IR that encodes both arithmetic semantics and data flow. Each IR instruction performs a scalar operation over field elements (including fixed-point encodings) and produces a single scalar result. Other instructions define a version of an array, a check, a value supplied by the prover with the checks that constrain it, or one output element of a whole array operator, which stays a placeholder until lowering. The tests run every compiled program on an IR interpreter against NumPy.

Circuit Synthesis and Backends. We provide a synthesis layer that lowers the IR into arithmetic circuits targeting two backends: (1) **Halo2** [4], a PLONK-based library with optimized gadgets for arithmetic and comparisons, in which the IR is laid out with lookup tables and a second phase for the verifier challenge; and (2) **Noir** [64], a DSL that compiles to ULTRAHONK circuits. The Noir backend emits a Noir program, mapping table reads and memory accesses to Noir's arrays.

Optimizations. To reduce circuit size and proving time, we apply standard IR optimizations prior to synthesis, including dead-code elimination, pattern-matching rewrites (algebraic simplification), and common-subexpression elimination. In addition, facts simplify the translation of operators: some operators are replaced by cheaper ones that are equal under the facts. The minimum of the result of `flatnonzero`, for instance, is its first set flag, so the IR scans for that flag and never builds the array, as for the example of Figure 3. After symbolic execution, a separate pass removes each assertion that its operands' definitions or an earlier assertion imply. Since it relies only on earlier assertions, two assertions never justify each other's removal.

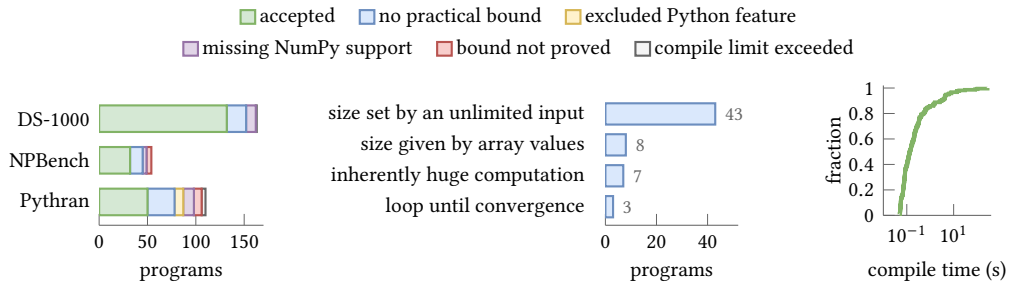
Arithmetic. An integer is a field element that is range-checked into the 64-bit window wherever its range is not already known. A real x is represented in fixed point as the integer round$(x \cdot 2^{32})$ [77, 91]. Products of reals, divisions and square roots take their result from the prover and check it. Each transcendental function has one fixed-point definition with a stated error bound, which the interpreter and both backends compute identically. Results therefore differ from IEEE 754 floating point, but the machine epsilon 2^{-32} is finer than the 2^{-23} of single precision, and the outputs of every accepted program differ from NumPy's double-precision results by less than 2×10^{-6}. Fixed point is common in zero-knowledge systems [29, 30], because emulating floating point in a circuit is expensive [34]. Every baseline, including the zkVM programs, uses the same fixed-point semantics, to which the correctness claim of Sec. 3.1 is relative.

8 Evaluation

Our evaluation answers three questions:

- **RQ1: Scope.** Which programs does the compiler accept, why does it reject the others, and what does accepting data-dependent shapes cost?
- **RQ2: Performance.** How do the compiled circuits compare in size and proving time with hand-written circuits and their direct translations, and with zkVMs, and where does the difference come from?
- **RQ3: Programming effort.** How much effort does writing an array program take with the compiler, compared with a zkVM and with a hand-written circuit?

For **RQ1**, we compile every program of three corpora of unmodified NumPy code and classify each rejection by its cause. For **RQ2**, we compare the circuit size and proving time of the compiled circuits with those of hand-written circuits on each backend and with zkVMs, on every program the system supports. For **RQ3**, we measure what efforts people take to write array programs with the compiler and with RISC0 in a user study.



(a) Outcome per corpus (b) Why no practical bound (c) Compile time

Fig. 9. **Which programs compile.** (a) Outcome for each of the 327 programs. (b) Why the 61 programs have no practical bound. (c) Compile time of the 214 accepted programs.

Environment. All experiments are conducted on a server with dual Intel(R) Xeon(R) Gold 6444Y CPUs 3.60GHz and 256GB RAM. Every prover runs with 8 threads, or 16 for the Groth16-wrapped zkVM proofs.

8.1 RQ1: Scope

We build our benchmark suite from three public corpora of NumPy code: the NumPy problems of DS-1000 [57], collected from Stack Overflow, and the scientific kernels of NPBench [100] and the Pythran benchmarks [38]. We keep every program that imports only NumPy and `math` and uses no exceptions, input and output, randomness, non-numeric data, resulting in 327 programs in total. Their function bodies stay *unchanged*. We declare each signature and the input domain that the upstream code implies. We compile those programs with Zinnia. Appendix D details the benchmark suite.

Accepted programs. As in Figure 9(a), the compiler accepts 214 of the 327 programs, and each accepted program matches NumPy on its test inputs. It accepts 81% of DS-1000, 59% of NPBench and 45% of Pythran. The accepted programs are not limited to fixed shapes. Some create an array whose shape depends on the input, such as the result of a mask, and the compiler derives its capacity, which a circuit language would require the programmer to write. 67 select or place elements according to the input, by a computed index, a mask or a sort, and the compiler chooses a construction for each such operator. To compare with an existing circuit compiler, we translate each program into Z#, the input language of CirC [65], statement by statement, keeping every shape and loop as in the source, and compile it to R1CS. CirC accepts only 117 programs.

Rejections. We classify each of the 113 rejections by reading the program and determining its worst case over the declared inputs in Figure 9(b). 70 lie outside the setting of Sec. 3.3. Of these 70, 61 have no practical bound or have no size limit under the declared inputs. The remaining 9 use Python features that our DSL excludes, such as sets. Of the remaining rejections, 25 need NumPy functions not yet implemented, 5 exceed compilation resource limits (300 s, 8 GB), and 13 have practical bounds the compiler fails to prove.

Cost of bounds. A circuit has a fixed size, so any circuit that accepts every valid input must reserve room for the largest one. This holds for circuits written by hand as much as for compiled ones. We measure how much more room the compiler reserves on the programs with data-dependent shapes. The largest size of an array is the largest number of elements it holds on any valid input. We compare the capacity that the compiler derives for each such array with its largest size. Except

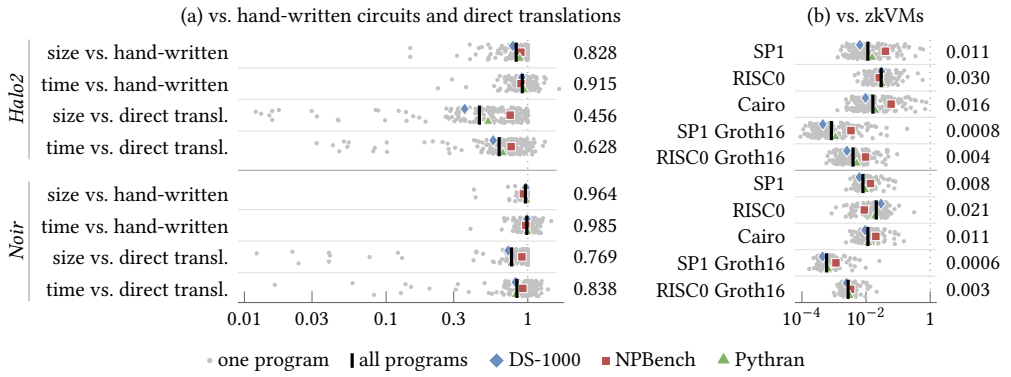


Fig. 10. **Compiled circuits over the baselines,** log scale, 172 programs. Values at right are geometric means. Below 1 the compiled circuit is smaller or faster. (a) Size and proving time over hand-written circuits and direct translations. (b) Proving time over zkVMs, without and with Groth16 wrapping.

for one program, the capacity of every array equals its largest size. Only one Pythran program has larger capacities. It reserves 256 elements for 6 of its 12 arrays, which hold at most 120, as we prove by hand and with an SMT solver. The compiler reserves this extra room because it does not derive that a matrix of pairwise differences is antisymmetric. This program's capacities are 36% larger than its largest sizes in total, and over all of these programs, the capacities are 1.0% larger on average. Over the same programs, the compiled circuits have 0.88× the cells of the hand-written circuits on Halo2 and 0.98× the gates on Noir.

Compile time. The median accepted program compiles in 0.13 s, and 207 of the 214 within ten seconds. The slowest, the ResNet basic block of NPBench, takes 282 s, most of it in symbolic execution of three convolutions written as Python loops.

Answer to RQ1: The compiler accepts 214 of 327 unmodified NumPy programs, including programs whose array shapes and accesses depend on the input. Most rejected programs lie outside our setting, and only 4% of all programs are rejected because the compiler fails to prove a bound.

8.2 RQ2: Performance

We use the 214 programs that Zinnia accepts in RQ1. Circuit compilers leave the capacities of such arrays to the programmer, and tools such as ezkl [29] target fixed-shape neural-network operators. We therefore write two baselines by hand for each program and backend. The *hand-written* circuit uses the constructions we know that a developer would choose to make it small. It is a best-effort baseline, and no construction is copied from the compiler's output. The *direct translation* translates each statement by the textbook algorithm of its NumPy function, such as a compare-and-swap network for a sort, and does not optimize across statements. Both share the compiler's input domains, integer widths and precision.

Measures. Circuit size, which does not vary between runs and determines most of the proving cost, is our primary measure. It is the number of advice and lookup cells on Halo2 and UltraHonk gates on Noir. Additionally, we measure proving time without setup. The six circuits of a program are timed one after another in one session, each by the median of three proofs, and we report geometric means of the ratios with 95% confidence intervals from bootstrapping the programs. We

leave out 36 programs that only copy data, such as `reshape`-only circuits, and 6 whose outputs follow from the input types alone, such as `a.shape`, because their circuits have little or nothing to compute. The remaining 172 programs form our comparison.

Hand-written circuits. We present our results in Figure 10(a). On average, the compiled circuits are smaller than the hand-written ones. On Halo2[2] they have 0.828× the cells and prove in 0.915× the time (95% CI [0.883, 0.943]). On Noir they have 0.964× the gates and prove in similar time, 0.985× ([0.963, 1.005]).

Direct translations. Compared with the direct translations, the compiled circuits have 0.456× the cells on Halo2 and 0.769× the gates on Noir, and prove in 0.628× (95% CI [0.565, 0.689]) and 0.838× ([0.769, 0.901]) the time. A direct translation lowers each operator by one fixed rule; the gap measures the compiler's combined benefit over direct translation, and the ablations below isolate individual mechanisms.

zkVMs. We compare with three zkVMs. SP1 6.8.1 [83] and RISC0 3.0.3 [73] prove Rust programs, written in the same fixed-point arithmetic as the circuits, and Stwo proves Cairo 2.20 programs. We run SP1, RISC0 and Cairo on all 172 programs. As shown in Figure 10(b), the compiled circuits prove faster than every zkVM, by 34 to 125× on average. By geometric mean, the Halo2 circuits prove in 0.011×, 0.030× and 0.016× the time of SP1, RISC0 and Cairo, and the Noir circuits in 0.008×, 0.021× and 0.011×. On every one of the 172 programs, compiled circuits are faster than SP1 and RISC0.

Cost at scale. We estimate the cost of proving at scale on an AWS r7iz.2xlarge instance, which has 8 virtual CPUs of the same processor generation as our server and costs $0.744 per hour. In serial proving, each input is proved on its own, one after another. Proving a million inputs of the median program then costs $32 with Zinnia's Noir circuit and $40 with its Halo2 circuit, against $1,413 with RISC0 and $4,913 with SP1. In batch proving, a zkVM proves many inputs in one execution, which spreads its fixed cost and the padding of its trace over all of them. Even if batching removed both entirely, a million inputs would still cost at least $236 with RISC0 and $230 with SP1. The Zinnia-compiled circuits can be batched as well, so comparing their serial cost with batched zkVMs favors the zkVMs.

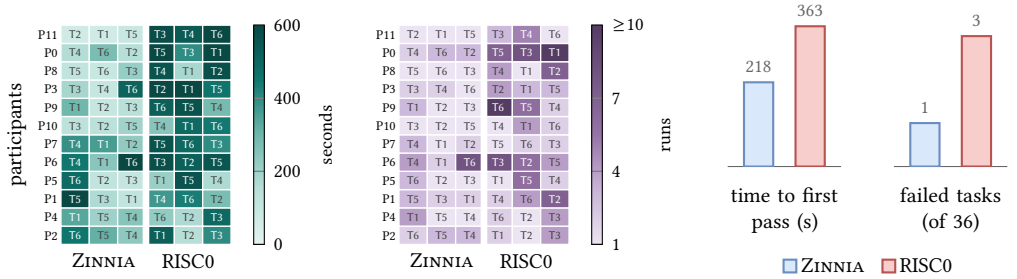
Where the gains come from. We disable the selection among several constructions per operator, the facts that selection uses, the array facts of symbolic execution, SMT reasoning about values, and the removal of checks that facts imply. On the programs whose operators have several constructions, the circuits then grow by 1.40× on Halo2 and 1.22× on Noir, and by up to 5.7× and 7.3×. Disabled one at a time, on the same programs, the circuits grow by 1.33× and 1.17× with one fixed construction per operator, 1.06× and 1.04× without the facts that selection uses, and 1.04× on both without the array facts. Leaving runtime-indexed reads and writes to the backend's own memory instead grows them by 1.03× and 1.10×. Appendix E gives per-program results.

Answer to RQ2: The compiled circuits are never larger than hand-written ones and on average smaller, less than half the size of direct translations on Halo2, and they prove 34 to 125× faster by geometric mean than zkVMs.

8.3 RQ3: Programming Effort

User study. To measure programming effort, we conduct a user study comparing Zinnia with RISC0. zkVMs are generally easy to use since they accept ordinary Rust programs by emulating them. The study compares Zinnia, whose Python-like syntax and NumPy API are familiar to data

[2]"On Halo2" and "on Noir" compare the circuits that Zinnia compiles to a backend with that backend's baselines.



(a) Active time per task (b) Test runs per task (c) Other measures

Fig. 11. **User study results.** (a) Active time and (b) test runs of every task, one row per participant. (c) Time to the first passing version and failed tasks, where lower is better.

scientists, with RISC0, which uses Rust syntax and the ndarray API. We recruited 12 participants: 6 postgraduate students and 6 software engineers, all claiming proficiency in both Rust and Python and none with experience in zero-knowledge proofs. Each participant completed 6 programming tasks (3 with Zinnia, 3 with RISC0) in a within-subjects (crossover) design, so no participant saw the same task in both languages. Task assignment was balanced by a Williams design [93]. Each task had a 10-minute time limit in an interactive web interface as an IDE providing instant feedback for compilation errors and a few test cases. We provided language and API manuals, allowed online searches, but disallowed coding agents. We measured task pass rate, number of attempts, and time. Additional details are in Appendix C.

Results. We present the results in Figure 11. All 12 participants needed less active time with Zinnia than with RISC0: 242 s against 443 s on average, a 45% reduction ($p = 0.002$ by a sign test with Holm correction). They reached a passing version sooner (218 s against 363 s), ran their tests half as often, and passed 35 of 36 tasks against 33 of 36 with RISC0.

Code size. We also measured code length on the 214 accepted programs in RQ1. The hand-written circuits are 4.8× as long as the NumPy programs on Halo2 and 3.7× on Noir, by geometric mean over lines without blank lines, comments and imports. Even the direct translations are 6.2× and 4.4× as long. The extra lines spell out what the compiler derives: the capacities of arrays and the constructions for data-dependent accesses.

Answer to RQ3: With Zinnia, every participant wrote array programs in less time than with RISC0, by 45% on average. The hand-written circuits are 3.7 to 4.8× as long as the NumPy programs they implement.

9 Related Work

Computational Models for ZKPs. The fixed-structure requirement of zk-SNARKs heavily constrains the computational model. Prior work extends zk-SNARKs to richer models in several ways. Some compiler frameworks translate high-level control flow into circuits by unrolling to a fixed bound [65, 68, 77, 88]. CirC [65] further uses SMT reasoning to resolve bounds for certain data-dependent control flows, but it accepts only 117 of our 327 programs (Sec. 8.1). Program synthesis-based approaches [46] synthesize an implementation with a user-provided high-level specification. Emulation-based systems prove execution by maintaining CPU state and verifying an instruction trace [9, 10, 16, 45, 73, 81, 83]. Proof folding and recursive SNARKs support incrementally

verifiable computation for long-running programs and unbounded loops [11, 17, 53]. Orthogonally, memory-checking arguments enable dynamic memory access by checking RAM consistency over a bounded trace [8–10, 66, 88]. However, "unbounded" systems like zkVMs and recursive proofs trade off generality against proving overhead, and typically require padding to ensure execution-trace obliviousness [74]. In contrast, Zinnia targets the common analytics programs where control flows and array operations are data dependent but *statically bounded*, and compiles them into fixed circuits via SMT-backed symbolic execution, treating a memory-checking argument as one construction among several.

Bounded Analysis and Symbolic Execution. Bounded model checking and symbolic execution explore the paths of a program with an SMT solver [5, 22, 63], mostly to find bugs, and worst-case execution time [92] and bound [39] analyses infer how often loops run. Zinnia instead uses the bounds and facts it proves to determine the capacities and constructions of a circuit.

Array Programming and Tensor Abstractions. Array-oriented analytics is commonly written as compositions of vectorized operators over tabular and multidimensional data (e.g., NumPy [40] and Pandas [62]). Languages like Futhark [41] and Lift [82] have formalized these patterns through size-dependent types and rewrite systems that enable high-performance GPU code generation. Dex [69] and Remora [79] check shapes through index sets and rank polymorphism, and static analysis infers the tensor shapes of TensorFlow programs [56]. While most such frameworks target CPU or GPU execution, recent work develops tensor abstractions tailored to cryptographic execution models, including for Fully Homomorphic Encryption [25, 54] and Secure Multiparty Computation [48]. Zinnia brings these high-level array-programming idioms to zk-SNARK compilation. It provides first-class multidimensional array primitives and transformations while ensuring that shape changes and dynamic indexing are compiled into fixed circuits.

ZKP Optimizing Compilers. A complementary line of work focuses on optimizing ZKP circuits. They apply algebraic rewrites [31], program synthesis, or protocol-specific transformations to reduce constraint counts or proving time [60]. Examples include IR-level optimizers [31, 65], lookup-enhanced transformations [60], and domain-specific optimizers for neural networks or SQL queries [18, 37]. They are complementary to Zinnia. xJsnark chooses a memory construction for each array that the programmer declares as memory [52], and ZKML picks an implementation for each layer of a neural network from a cost model [18]. Both choose per array or per layer, whereas Zinnia selects the constructions of all operators together and tracks which construction enforces each fact that another relies on. Recent work removes range and sign checks implied by other constraints from compiled neural-network circuits [97], whereas Zinnia derives facts from the source program before any circuit exists.

Zero-knowledge Program Analysis. A growing body of work develops analysis and verification tools for ZKP programs and circuits, including detection of functional bugs [96], security vulnerabilities [67, 86, 90], and formal verification [59]. They analyze an existing circuit or DSL program and apply to the circuits that Zinnia generates as well.

10 Conclusion

Zinnia compiles array programs whose shapes, accesses and control depend on the data into fixed zk-SNARK circuits, choosing the constructions of all operators together by the facts it derives. Its circuits are smaller on average than hand-written ones, and its proofs are faster than with zkVMs.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. 2025. Finch: Sparse and Structured Tensor Programming with Control Flow. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (April 2025), 1042–1072. doi:10.1145/3720473

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1792–1803. doi:10.14778/2824032.2824076

[4] axiom-crypto. n.d.. halo2-lib. https://github.com/axiom-crypto/halo2-lib Accessed 2026-10-11.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (May 2018), 1–39. doi:10.1145/3182657

[6] K. E. Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring) (AFIPS '68 (Spring))*. ACM Press, 307. doi:10.1145/1468075.1468121

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

[8] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. 2013. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: extended abstract. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science (ITCS '13)*. ACM, 401–414. doi:10.1145/2422436.2422481

[9] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *Advances in Cryptology – CRYPTO 2013*, Ran Canetti and Juan A. Garay (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 90–108.

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

[11] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2017. Scalable Zero Knowledge Via Cycles of Elliptic Curves. *Algorithmica* 79, 4 (01 Dec 2017), 1102–1160. doi:10.1007/s00453-016-0221-0

[12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2012. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*. ACM, 326–349. doi:10.1145/2090236.2090263

[13] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. *vZ* - An Optimizing SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 194–199.

[14] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. 1994. Checking the correctness of memories. *Algorithmica* 12, 2 (01 Sep 1994), 225–244. doi:10.1007/BF01185212

[15] 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 - STOC '88 (STOC '88)*. ACM Press, 103–112. doi:10.1145/62212.62222

[16] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. 2018. Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution. In *Advances in Cryptology – ASIACRYPT 2018*, Thomas Peyrin and Steven Galbraith (Eds.). Springer International Publishing, Cham, 595–626.

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

[18] 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 (EuroSys '24)*. ACM, 560–574. doi:10.1145/3627703.3650088

[19] Min Chen, Shiwen Mao, and Yunhao Liu. 2014. Big Data: A Survey. *Mobile Networks and Applications* 19, 2 (01 Apr 2014), 171–209. doi:10.1007/s11036-013-0489-0

[20] Thomas Chen, Hui Lu, Teeramet Kunpittaya, and Alan Luo. 2023. A Review of zk-SNARKs. arXiv:2202.06877 [cs.CR] https://arxiv.org/abs/2202.06877

[21] 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, Paper 2021/651. https://eprint.iacr.org/2021/651

[22] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Kurt Jensen and Andreas Podelski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 168–176.

[23] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77 (POPL '77)*. ACM Press, 238–252. doi:10.1145/512950.512973

[24] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '11)*. ACM, 105–118. doi:10.1145/1926385.1926399

[25] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. ACM, 142–156. doi:10.1145/3314221.3314628

[26] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[27] 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)*. IEEE, 1084–1091. doi:10.1109/cybermatics_2018.2018.00199

[28] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. 2008. Generalized instruction selection using SSA -graphs. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES08)*. ACM, 31–40. doi:10.1145/1375657.1375663

[29] EZKL. n.d.. EZKL. https://ezkl.xyz/ Accessed 2026-10-11.

[30] 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, Paper 2021/087. https://eprint.iacr.org/2021/087

[31] Matthew Fredrikson and Benjamin Livshits. 2014. ZØ: An Optimizing Distributing Zero-Knowledge Compiler. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 909–924. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/fredrikson

[32] Rusins Freivalds. 1977. Probabilistic Machines Can Use Less Running Time. In *Information Processing, Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977*, Bruce Gilchrist (Ed.). North-Holland, 839–842.

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

[34] Sanjam Garg, Abhishek Jain, Zhengzhong Jin, and Yinuo Zhang. 2022. Succinct Zero Knowledge for Floating Point Computations. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. ACM, 1203–1216. doi:10.1145/3548606.3560653

[35] Goetz Graefe. 1993. Query evaluation techniques for large databases. *Comput. Surveys* 25, 2 (June 1993), 73–169. doi:10.1145/152610.152611

[36] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 305–326.

[37] Binbin Gu, Juncheng Fang, and Faisal Nawab. 2025. PoneglyphDB: Efficient Non-interactive Zero-Knowledge Proofs for Arbitrary SQL-Query Verification. *Proceedings of the ACM on Management of Data* 3, 1 (Feb. 2025), 1–27. doi:10.1145/3709713

[38] Serge Guelton, Pierrick Brunet, Mehdi Amini, Adrien Merlini, Xavier Corbillon, and Alan Raynaud. 2015. Pythran: enabling static optimization of scientific Python programs. *Computational Science & Discovery* 8, 1 (March 2015), 014001. doi:10.1088/1749-4680/8/1/014001

[39] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL09)*. ACM, 127–139. doi:10.1145/1480881.1480898

[40] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H.

van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (September 2020), 357–362. doi:10.1038/s41586-020-2649-2

[41] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, 556–571. doi:10.1145/3062341.3062354

[42] Gabriel Hjort Blindell. 2016. *Instruction Selection*. Springer International Publishing. doi:10.1007/978-3-319-34019-7

[43] Ross Ihaka and Robert Gentleman. 1996. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics* 5, 3 (September 1996), 299–314. doi:10.1080/10618600.1996.10474713

[44] Kenneth E. Iverson. 2007. *Notation as a tool of thought*. ACM, 1979. doi:10.1145/1283920.1283935

[45] Kunming Jiang, Fraser Brown, and Riad S. Wahby. 2025. CoBBL: Dynamic Constraint Generation for SNARKs. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3347–3363. doi:10.1109/sp61157.2025.00200

[46] Kunming Jiang, Devora Chait-Roth, Zachary DeStefano, Michael Walfish, and Thomas Wies. 2023. Less is more: refinement proofs for probabilistic proofs. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1112–1129. doi:10.1109/sp46215.2023.10179393

[47] Richard M. Karp. 1972. *Reducibility among Combinatorial Problems*. Springer US, Boston, MA, 85–103. doi:10.1007/978-1-4684-2001-2_9

[48] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. ACM, 1575–1590. doi:10.1145/3372297.3417872

[49] Gary A. Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73 (POPL '73)*. ACM Press, 194–206. doi:10.1145/512927.512945

[50] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (July 1976), 385–394. doi:10.1145/360248.360252

[51] David Ryan Koes and Seth Copen Goldstein. 2008. Near-optimal instruction selection on dags. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization (CGO '08)*. ACM, 45–54. doi:10.1145/1356058.1356065

[52] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. 2018. xJsnark: A Framework for Efficient Verifiable Computation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 944–961. doi:10.1109/sp.2018.00018

[53] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *Advances in Cryptology – CRYPTO 2022*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer Nature Switzerland, Cham, 359–388.

[54] Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. 2024. A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption. *Proceedings of the ACM on Programming Languages* 8, PLDI (June 2024), 126–150. doi:10.1145/3656382

[55] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. *ACM SIGPLAN Notices* 47, 6 (June 2012), 193–204. doi:10.1145/2345156.2254088

[56] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:29. doi:10.4230/LIPIcs.ECOOP.2020.15

[57] 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 (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 18319–18345. https://proceedings.mlr.press/v202/lai23b.html

[58] Xiling Li, Chenkai Weng, Yongxin Xu, Xiao Wang, and Jennie Rogers. 2023. ZKSQL: Verifiable and Efficient Query Evaluation with Zero-Knowledge Proofs. *Proceedings of the VLDB Endowment* 16, 8 (April 2023), 1804–1816. doi:10.14778/3594512.3594513

[59] 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)*. IEEE, 1741–1759. doi:10.1109/sp54263.2024.00078

[60] Junrui Liu, Jiaxin Song, Yanning Chen, Hanzhi Liu, Hongbo Wen, Luke Pearson, Yanju Chen, and Yu Feng. 2025. Tabby: A Synthesis-Aided Compiler for High-Performance Zero-Knowledge Proof Circuits. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (Oct. 2025), 1671–1697. doi:10.1145/3763110

[61] Tianyi Liu, Xiang Xie, and Yupeng Zhang. 2021. zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. ACM, 2968–2985. doi:10.1145/3460120.3485379

[62] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference (SciPy)*. SciPy, 56–61. doi:10.25080/majora-92bf1922-00a

[63] Florian Merz, Stephan Falke, and Carsten Sinz. 2012. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In *Verified Software: Theories, Tools, Experiments*, Rajeev Joshi, Peter Müller, and Andreas Podelski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 146–161.

[64] Noir. n.d.. Noir. https://noir-lang.org/ Accessed 2026-10-11.

[65] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. 2022. CirC: Compiler infrastructure for proof systems, software verification, and more. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2248–2266. doi:10.1109/sp46214.2022.9833782

[66] Alex Ozdemir, Evan Laufer, and Dan Boneh. 2025. Volatile and Persistent Memory for zkSNARKs via Algebraic Interactive Proofs. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3309–3327. doi:10.1109/sp61157.2025.00054

[67] 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 (June 2023), 1510–1532. doi:10.1145/3591282

[68] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2016. Pinocchio: nearly practical verifiable computation. *Commun. ACM* 59, 2 (Jan. 2016), 103–112. doi:10.1145/2856449

[69] Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021), 1–29. doi:10.1145/3473593

[70] E. Patterson, R. McBurney, H. Schmidt, I. Baldini, A. Mojsilovic, and K. R. Varshney. 2017. Dataflow representation of data analyses: Toward a platform for collaborative data science. *IBM Journal of Research and Development* 61, 6 (Nov. 2017), 9:1–9:13. doi:10.1147/jrd.2017.2736278

[71] Luke Pearson, Joshua Fitzgerald, Héctor Masip, Marta Bellés-Muñoz, and Jose Luis Muñoz-Tapia. 2022. PlonKup: Reconciling PlonK with plookup. Cryptology ePrint Archive, Paper 2022/086. https://eprint.iacr.org/2022/086

[72] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems* 22, 1 (Jan. 2000), 1–44. doi:10.1145/345099.345100

[73] RISC Zero. n.d.. RISC Zero. https://github.com/risc0/risc0 Accessed 2026-10-11.

[74] RISC Zero. n.d.. STARK by Hand. https://dev.risczero.com/proof-system/stark-by-hand Accessed 2026-10-11.

[75] J. T. Schwartz. 1980. Fast Probabilistic Algorithms for Verification of Polynomial Identities. *J. ACM* 27, 4 (Oct. 1980), 701–717. doi:10.1145/322217.322225

[76] scipr-lab. n.d.. libsnark. https://github.com/scipr-lab/libsnark Accessed 2026-10-11.

[77] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. 2012. Taking Proof-Based Verified Computation a Few Steps Closer to Practicality. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 253–268. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/setty

[78] Mohammad Shahnawaz and Manish Kumar. 2025. A Comprehensive Survey on Big Data Analytics: Characteristics, Tools and Techniques. *Comput. Surveys* 57, 8 (March 2025), 1–33. doi:10.1145/3718364

[79] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–46.

[80] Tobin South, Alexander Camuto, Shrey Jain, Shayla Nguyen, Robert Mahari, Christian Paquin, Jason Morton, and Alex 'Sandy' Pentland. 2024. Verifiable evaluations of machine learning models using zkSNARKs. arXiv:2402.02675 [cs.LG] https://arxiv.org/abs/2402.02675

[81] StarkWare. n.d.. Cairo. https://github.com/starkware-libs/cairo Accessed 2026-10-11.

[82] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 74–85. doi:10.1109/cgo.2017.7863730

[83] Succinct Labs. n.d.. SP1. https://github.com/succinctlabs/sp1 Accessed 2026-10-11.

[84] The Electric Coin Company and the Halo 2 developers. n.d.. Halo2. https://github.com/zcash/halo2 Accessed 2026-10-11.

[85] Quoc-Cuong To, Juan Soto, and Volker Markl. 2018. A survey of state management in big data processing systems. *The VLDB Journal* 27, 6 (01 Dec 2018), 847–872. doi:10.1007/s00778-018-0514-9

[86] Trail of Bits. n.d.. Circomspect. https://github.com/trailofbits/circomspect Accessed 2026-10-11.

[87] A. M. Turing. 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42, 1 (1937), 230–265. doi:10.1112/plms/s2-42.1.230

[88] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. 2015. Efficient RAM and control flow in verifiable outsourced computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society. https://www.ndss-symposium.org/ndss2015/efficient-ram-and-control-flow-verifiable-outsourced-computation

[89] Abraham Waksman. 1968. A Permutation Network. *J. ACM* 15, 1 (Jan. 1968), 159–163. doi:10.1145/321439.321449

[90] 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)*. USENIX Association, Philadelphia, PA, 1471–1487. https://www.usenix.org/conference/usenixsecurity24/presentation/wen

[91] 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)*. USENIX Association, 501–518. https://www.usenix.org/conference/usenixsecurity21/presentation/weng

[92] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems* 7, 3 (April 2008), 1–53. doi:10.1145/1347375.1347389

[93] EJ Williams. 1949. Experimental Designs Balanced for the Estimation of Residual Effects of Treatments. *Australian Journal of Scientific Research Series A: Physical Sciences* 2, 2 (June 1949), 149–168. doi:10.1071/ch9490149

[94] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 1–29. doi:10.1145/3434304

[95] Hao Wu, Changzheng Wei, Yanhao Wang, Li Lin, Yilong Leng, Shiyu He, Minghao Zhao, Hanghang Wu, Ying Yan, and Aoying Zhou. 2025. Zero-Knowledge Verifiable Graph Query Evaluation via Expansion-Centric Operator Decomposition. arXiv:2507.00427 [cs.DB] https://arxiv.org/abs/2507.00427

[96] Dongwei Xiao, Zhibo Liu, Yiteng Peng, and Shuai Wang. 2025. MTZK: Testing and Exploring Bugs in Zero-Knowledge (ZK) Compilers. In *Proceedings 2025 Network and Distributed System Security Symposium (NDSS 2025)*. Internet Society. doi:10.14722/ndss.2025.230530

[97] Zhantong Xue, Pingchuan Ma, Zhaoyu Wang, Yuguang Zhou, Huaijin Wang, and Shuai Wang. 2026. Sound Debloating of Redundant Checks in Zero-Knowledge Machine-Learning Circuits. arXiv:2609.10149 [cs.CR] https://arxiv.org/abs/2609.10149

[98] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268. https://proceedings.mlsys.org/paper_files/paper/2021/file/cc427d934a7f6c0663e5923f49eba531-Paper.pdf

[99] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 863–880. doi:10.1109/sp.2017.43

[100] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. 2021. NPBench: a benchmarking suite for high-performance NumPy. In *Proceedings of the ACM International Conference on Supercomputing (ICS '21)*. ACM, 63–74. doi:10.1145/3447818.3460360

A Symbolic Execution Rules and Construction Library

This appendix gives the rules of symbolic execution, the bounds of shape operators, and the construction library of Sec. 5.3, in the notation of Figure 4 and Figure 7. The rules omit the typing of scalar operators, which Op covers.

A.1 The Core Language

The rules cover the following core of the language. In it, x ranges over variables, c over literals, f over the operators that have a specification (arithmetic, comparisons, NumPy functions, slices and masks), and h over the functions of the program:

$$
\begin{aligned}
e &::= x \mid c \mid f(e_1, \ldots, e_n) \mid e[e] \mid e \texttt{ if } e \texttt{ else } e \mid h(e_1, \ldots, e_n) \\
s &::= x = e \mid x[e] = e \mid \texttt{assert } e \mid \texttt{pass} \mid s\texttt{; } s \mid \texttt{if } e\texttt{: } s \texttt{ else: } s \mid \texttt{break} \mid \texttt{continue} \mid \texttt{return } e \\
&\quad \mid \texttt{ for } x \texttt{ in range}(e, e)\texttt{: } s \texttt{ else: } s \mid \texttt{for } x \texttt{ in } e\texttt{: } s \texttt{ else: } s \mid \texttt{while } e\texttt{: } s
\end{aligned}
$$

Other forms, such as `elif`, augmented assignments and comprehensions, reduce to these. `and` and `or` evaluate both operands.

A.2 The Rules

The rules use the notation of Sec. 4.2, extended as follows. The environment Γ holds four reserved variables: the value returned so far rv, and the conditions rd, bk and ct under which the function has returned and the innermost loop has been left by `break` or `continue`; $ex(\Gamma) = \Gamma(rd) \vee \Gamma(bk) \vee \Gamma(ct)$. Each bound variable x carries the condition $\beta(x)$ under which it has a value. $\sigma[x \Leftarrow v]$ is a guarded assignment: it binds x to v where the path condition p holds, through $v' = \texttt{select}(p, v, \Gamma(x))$ when x is already bound, and sets $\beta(x)$ to $\beta(x) \vee p$. $wrap_p(i, n)$ is the index the IR uses for i on an axis of runtime extent n ($i + n$ for $i < 0$, and 0 off the path), and $\text{write}_p(a, k, v)$ is a with cell k set to v where p holds. In If, $\Gamma_1 \sqcup_c \Gamma_2$ binds each variable bound in both branches to the select of the two values under c, with the join of their types; a variable bound in only one branch and not before is dropped. $\sigma \vdash^j W \dashv \sigma'$ executes the loop $W = \texttt{while } e\texttt{: } s$ after j iterations, L_{loop} (1000 by default) bounds j, and L_{call} (16 by default) bounds the depth δ of inlined calls. $\eta(\sigma)$ is the set of functions whose inlined bodies enclose the current statement, and $\mathbf{0}_h$ is the zero value of the return type of h. While-Cap applies with the cap J to a guard $j > 0$ on a variable of an enclosing `for` loop that the body only decrements, and While-Range to a guard $i < e$ whose counter the body only increments by a literal; elsewhere $J = \infty$. A `for` loop over an array is For-Range over its first axis, and a range with a step is handled alike. A program starts from $\langle [\vec{x} \mapsto \vec{u}],\ F_D,\ \texttt{true},\ I_D \rangle$, where I_D asserts the input domain and F_D states the preconditions. If, For-Range, While-Step and While-Exit refine the rules of the same names in Figure 4.

(a) Expressions

$$\textsc{Var}\ \frac{\Gamma(x) = v : \tau \qquad \sigma \vdash \text{require } \beta(x) \dashv \sigma'}{\sigma \vdash x \Downarrow v : \tau \dashv \sigma'} \qquad \textsc{Const}\ \frac{c \text{ a literal of type } \tau}{\sigma \vdash c \Downarrow c : \tau \dashv \sigma}$$

$$\textsc{Read}\ \frac{\sigma \vdash e_1, e_2 \Downarrow a, i : \langle s;\ \vec{d} \mid L\rangle, \texttt{Int} \dashv \sigma_1 \qquad n = |a|_1 \qquad \sigma_1 \vdash \text{require } -n \le i < n \dashv \sigma_2 \qquad k, r \text{ fresh}}{\sigma \vdash e_1[e_2] \Downarrow r : s \dashv \sigma_2 \oplus_p r \in rng(a) \lhd k = wrap_p(i, n) \lhd r = \text{read}(a, k)}$$

$$\textsc{Cond}\ \frac{\begin{array}{c}\sigma \vdash e_0 \Downarrow c \dashv \sigma_0 \\ c \notin \{\texttt{true}, \texttt{false}\} \qquad \sigma_0[p \wedge c] \vdash e_1 \Downarrow v_1 : \tau_1 \dashv \sigma_1 \qquad \sigma_1[p \wedge \neg c] \vdash e_2 \Downarrow v_2 : \tau_2 \dashv \sigma_2 \qquad v \text{ fresh}\end{array}}{\sigma \vdash e_1 \texttt{ if } e_0 \texttt{ else } e_2 \Downarrow v : \tau_1 \sqcup \tau_2 \dashv \sigma_2[p] \lhd v = \texttt{select}(c, v_1, v_2)}$$

$$\textsc{Call}\ \frac{\begin{array}{c}\texttt{def } h(\vec{x})\texttt{:}\ s_h \qquad \sigma \vdash \vec{e} \Downarrow \vec{v} \dashv \sigma_0 \qquad \delta(\sigma_0) < L_{call} \qquad h \notin \eta(\sigma_0) \text{ or } F_0 \not\models_p \neg p \\ \langle [\vec{x} \mapsto \vec{v},\ rd, bk, ct \mapsto \texttt{false}],\ F_0, p_0, I_0\rangle \vdash s_h \dashv \sigma_1 \text{ at depth } \delta(\sigma_0) + 1 \qquad \Gamma_1(rd) = \texttt{true} \text{ or } s_h \text{ has no } \texttt{return}\end{array}}{\sigma \vdash h(\vec{e}) \Downarrow \Gamma_1(rv) \dashv \langle \Gamma_0, F_1, p_0, I_1\rangle}$$

$$\textsc{Call-Exit}\ \frac{\sigma \vdash \vec{e} \Downarrow \vec{v} \dashv \sigma_0 \qquad h \in \eta(\sigma_0) \qquad F_0 \models_p \neg p}{\sigma \vdash h(\vec{e}) \Downarrow \mathbf{0}_h \dashv \sigma_0 \lhd \text{assert}_p\ \neg p}$$

(b) Statements

$$\textsc{Assign}\ \frac{\sigma \vdash e \Downarrow v : \tau \dashv \sigma_0}{\sigma \vdash x = e \dashv \sigma_0[x \Leftarrow v]} \qquad \textsc{Assert}\ \frac{\sigma \vdash e \Downarrow c \dashv \sigma_0 \qquad \sigma_0 \vdash \text{require } c \dashv \sigma_1}{\sigma \vdash \texttt{assert } e \dashv \sigma_1}$$

$$\textsc{Seq}\ \frac{\sigma \vdash s_1 \dashv \sigma_1 \qquad \sigma_1[p \wedge \neg ex(\Gamma_1)] \vdash s_2 \dashv \sigma_2}{\sigma \vdash s_1;\ s_2 \dashv \sigma_2[p]}$$

$$\textsc{Write}\ \frac{\sigma \vdash x, e_1, e_2 \Downarrow a, i, v : \langle s;\ \vec{d} \mid L\rangle, \texttt{Int}, s \dashv \sigma_1 \qquad n = |a|_1 \qquad \sigma_1 \vdash \text{require } -n \le i < n \dashv \sigma_2 \qquad k, a' \text{ fresh}}{\sigma \vdash x[e_1] = e_2 \dashv \sigma_2[x \mapsto a'] \oplus_p a' \triangleright \mathcal{S}_a \sqcup rng(v) \lhd k = wrap_p(i, n) \lhd a' = \text{write}_p(a, k, v)}$$

$$\textsc{If}\ \frac{\begin{array}{c}\sigma \vdash e \Downarrow c \dashv \sigma_0 \qquad c \notin \{\texttt{true}, \texttt{false}\} \\ \sigma_0[p \wedge c] \vdash s_1 \dashv \sigma_1 \qquad \sigma_0[p \wedge \neg c] \vdash s_2 \dashv \sigma_2\end{array}}{\sigma \vdash \texttt{if } e\texttt{:}\ s_1 \texttt{ else:}\ s_2 \dashv \langle \Gamma_1 \sqcup_c \Gamma_2,\ F_1 \cup F_2,\ p,\ I_1 \cup I_2\rangle} \qquad \textsc{If-True}\ \frac{\begin{array}{c}\sigma \vdash e \Downarrow \texttt{true} \dashv \sigma_0 \\ \sigma_0 \vdash s_1 \dashv \sigma_1\end{array}}{\sigma \vdash \texttt{if } e\texttt{:}\ s_1 \texttt{ else:}\ s_2 \dashv \sigma_1}$$

$$\textsc{Return}\ \frac{\sigma \vdash e \Downarrow v \dashv \sigma_0 \qquad d \text{ fresh}}{\sigma \vdash \texttt{return } e \dashv \sigma_0[rv \Leftarrow v][rd \mapsto d] \lhd d = \Gamma_0(rd) \vee p} \qquad \textsc{Break}\ \frac{b \text{ fresh}}{\sigma \vdash \texttt{break} \dashv \sigma[bk \mapsto b] \lhd b = \Gamma(bk) \vee p}$$

$$\textsc{Continue}\ \frac{b \text{ fresh}}{\sigma \vdash \texttt{continue} \dashv \sigma[ct \mapsto b] \lhd b = \Gamma(ct) \vee p}$$

(c) Loops

$$\textsc{For-Range}\ \frac{\begin{array}{c}\sigma \vdash e_1, e_2 \Downarrow a, b \dashv \sigma_0 \qquad K = \max(0, \max_{F_0, p}(b - a)) \qquad \sigma'_0 = \sigma_0[bk \mapsto \texttt{false}] \\ \sigma'_k[ct \mapsto \texttt{false}][p_k][x \Leftarrow a + k] \vdash s \dashv \sigma'_{k+1} \quad (0 \le k < K) \\ p_k = p \wedge a + k < b \wedge \neg\Gamma'_k(bk) \wedge \neg\Gamma'_k(rd) \\ \sigma'_K[p] \vdash \text{require } a + K \ge b \dashv \sigma'' \qquad \sigma''[p \wedge \neg\Gamma''(bk) \wedge \neg\Gamma''(rd)] \vdash s' \dashv \sigma'''\end{array}}{\sigma \vdash \texttt{for } x \texttt{ in range}(e_1, e_2)\texttt{:}\ s \texttt{ else:}\ s' \dashv \sigma'''[p][bk \mapsto \Gamma_0(bk)][ct \mapsto \Gamma_0(ct)]}$$

$$\textsc{While}\ \frac{\sigma[bk \mapsto \texttt{false}] \vdash^0 W \dashv \sigma_1}{\sigma \vdash W \dashv \sigma_1[p][bk \mapsto \Gamma(bk)][ct \mapsto \Gamma(ct)]} \qquad \textsc{While-Exit}\ \frac{\begin{array}{c}\sigma \vdash e \Downarrow c \dashv \sigma_0 \\ F_0 \models_p \neg c\end{array}}{\sigma \vdash^j W \dashv \sigma_0}$$

$$\textsc{While-Step}\ \frac{\begin{array}{c}\sigma \vdash e \Downarrow c \dashv \sigma_0 \qquad F_0 \not\models_p \neg c \qquad j < L_{loop} \qquad j < J \\ \sigma_0[ct \mapsto \texttt{false}][p \wedge c] \vdash s \dashv \sigma_1 \qquad \sigma_1[p \wedge c \wedge \neg\Gamma_1(bk) \wedge \neg\Gamma_1(rd)] \vdash^{j+1} W \dashv \sigma_2\end{array}}{\sigma \vdash^j W \dashv \sigma_2[p]}$$

$$\textsc{While-Cap}\ \frac{\sigma \vdash e \Downarrow c \dashv \sigma_0 \qquad j = J}{\sigma \vdash^j W \dashv \sigma_0 \oplus_p \neg c \lhd \text{assert}_p\ \neg c}$$

$$\textsc{While-Range}\ \frac{W \text{ is } \texttt{while } i < e\texttt{:}\ s \text{ with the counted form for } t \qquad \sigma \vdash \texttt{for } y \texttt{ in range}(\Gamma(i), e, t)\texttt{:}\ s \dashv \sigma' \qquad y \text{ fresh}}{\sigma \vdash W \dashv \sigma'}$$

The join of types. The join $\tau_1 \sqcup \tau_2$ is defined as follows.

- Scalars join by numeric promotion along `Bool` < `Int` < `Real`.
- Two lists of the same length join element by element, and so do two tuples of the same length. Two lists or tuples of numbers of different lengths are converted into bounded-dynamic arrays $\langle s;\ n \mid n \rangle$ and join as arrays.
- Two static arrays of the same shape keep it, $\langle s;\ \vec{d} \rangle \sqcup \langle s';\ \vec{d} \rangle = \langle s \sqcup s';\ \vec{d} \rangle$.
- Two one-dimensional static arrays of different lengths, and two arrays of the same rank of which one is bounded-dynamic, join by the rule of Figure 4(a), in which a static array counts as a bounded-dynamic one whose capacity is the product of its extents, and $s \sqcup_a s'$ is `Real` if either is `Real` and `Int` otherwise. The runtime shape of the result is selected by the branch condition.
- An array that is empty at compile time, such as `np.array([])`, joins with an array of the same element family as a bounded-dynamic array of the rank of the other side whose bounds are all 0.

The join is undefined otherwise, for example for arrays of different ranks, for static arrays of different shapes with several axes, and for an array and a number. Where the join of a variable is undefined, $\Gamma_1 \sqcup_c \Gamma_2$ binds the variable to a placeholder, and the program is rejected where the variable is read. An undefined join of the values that a function returns is rejected at the join.

A.3 Bounds of Shape Operators

The specifications below show how shape operators transform the bounds of bounded-dynamic arrays (Sec. 4.3), in the form of Figure 4(c), with the postconditions that concern only values omitted. In them, x has the type $\langle s;\ d_1, \ldots, d_k \mid L \rangle$ and y the type $\langle s';\ d'_1, \ldots, d'_k \mid L' \rangle$, a static array counts as a bounded-dynamic one whose capacity is the product of its extents, axes and permutations are compile-time constants, and $D = \prod_i d_i$.

$$
\begin{aligned}
\{\top\}\ & \texttt{transpose}(x, \pi) : \langle s;\ d_{\pi(1)}, \ldots, d_{\pi(k)} \mid L \rangle \\
\{\ell_x = t_1 \cdots t_m\}\ & \texttt{reshape}(x, (t_1, \ldots, t_m)) : \langle s;\ t_1, \ldots, t_m \rangle \\
& \quad \text{if every } t_j \text{ is constant and } t_1 \cdots t_m = D \\
\{\ell_x = u_1 \cdots u_m \wedge \vec{u} \le \vec{e}\}\ & \texttt{reshape}(x, (u_1, \ldots, u_m)) : \langle s;\ e_1, \ldots, e_m \mid D \rangle \\
& \quad \text{with } e_j = u_j \text{ for a constant } u_j \text{ and } e_j = D / \textstyle\prod_{\text{constant } u_q} u_q \text{ otherwise} \\
\{\textstyle\bigwedge_{i \neq a} |x|_i = |y|_i\}\ & \texttt{concatenate}((x, y), a) : \langle s \sqcup_a s';\ d_1, \ldots, d_a + d'_a, \ldots, d_k \mid L + L' \rangle \\
& \quad \text{if } d_i = d'_i \text{ for } i \neq a, \quad \{\ell_r = \ell_x + \ell_y\} \text{ for } k = 1 \\
\{\top\}\ & \texttt{sum}(x, a) : \langle s;\ d_1, \ldots, d_{a-1}, d_{a+1}, \ldots, d_k \mid \textstyle\prod_{i \neq a} d_i \rangle \\
& \quad \text{likewise } \texttt{prod}, \texttt{any}, \texttt{all}, \text{ and, under } \{|x|_a > 0\}, \\
& \quad \texttt{amax}, \texttt{amin}, \texttt{argmax}, \texttt{argmin} \\
\{\ell_x > 0\}\ & \texttt{amax}(x) : s, \ \text{likewise } \texttt{amin}, \texttt{argmax}, \texttt{argmin}, \texttt{mean} \\
\{\top\}\ & x[e_1 : e_2] \text{ along axis } a : \langle s;\ \vec{d} \mid L \rangle \\
\{\top\}\ & x[h : \langle \texttt{Bool};\ \vec{d} \mid L \rangle] : \langle s;\ L \mid L \rangle \quad \{0 \le \ell_r \le L\} \\
\{\textstyle\bigwedge_i |x|_i = |y|_i\}\ & x \odot y : \langle s \sqcup s';\ \min(d_1, d'_1), \ldots, \min(d_k, d'_k) \mid \min(L, L') \rangle
\end{aligned}
$$

In the last line, $\odot$ is an element-wise operator without broadcasting. A precondition that the facts do not prove is asserted in the IR, and a bound that is not a compile-time constant is rejected.

A.4 The Construction Library

The library has 45 constructions in seven families, written as in Figure 7: $\{Pre\}\ f \rightsquigarrow C\ \{Post\}$, hatted values come from the prover, γ is the verifier challenge, and facts are in teal. A precondition in square brackets is optional and only narrows checks. $E(r)$ is the one-hot vector of a position r and $T(r) = ([r < k])_k$ its thermometer. (Halo2) or (Noir) marks a construction that exists on one target only.

(a) Sorting and searching

Here $(s, p) = sort(x)$ computes the sorted values s and the stable order p of x at the ranks that the program uses, for `np.sort`, `np.argsort`, `np.median`, `np.percentile`, and payloads $y[p]$, and $rng(x)$ is the range of its elements.

SORTING-NETWORK $\{[rng(x)]\}$ $(s, p) = sort(x) \rightsquigarrow$
a network of compare-exchanges (Batcher's merge exchange, or adjacent swaps)
on x_j, (x_j, j) or $x_j 2^b + j$, reduced to the comparators that the used outputs depend on

RANK-SELECTORS $\{[rng(x)]\}$ $(s, p) = sort(x) \rightsquigarrow$
$c_{jk} = [(x_j, j) < (x_k, k)]$, $e_{rj} = [\sum_k c_{kj} = r]$, $s_r = \sum_j e_{rj} x_j$, $p_r = \sum_j e_{rj} j$

SORT-HINT $\{[rng(x)]\}$ $p = argsort(x) \rightsquigarrow$
$p = \hat{p}$, $0 \le p_r < n$, $(x[p_r], p_r) < (x[p_{r+1}], p_{r+1})$ $\{p = argsort(x)\}$

SELECTED-RANKS $\{[rng(x)]\}$ $s_r = sort(x)_r \rightsquigarrow$
$s_r = \hat{s}_r$, $\#\{j : x_j < s_r\} \le r < \#\{j : x_j \le s_r\}$ for an isolated rank r,
a run of ranks checked increasing and placed by one count,
$s_0 \le x_j$ for all j and $\prod_j (x_j - s_0) = 0$ for rank 0, likewise for rank $n - 1$

ROUTED-SORT $\{[rng(x)]\}$ $(s, p) = sort(x) \rightsquigarrow$ (Halo2)
$K_j = x_j 2^b + j$, $\hat{w} \in \{0, 1\}^{W(n)}$, $K' = Waksman_{\hat{w}}(K)$, $K'_r < K'_{r+1}$, (s_r, p_r) unpacked from K'_r,
$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1$

SORT-PRODUCT $\{\gamma\}$ $s = sort(K) \rightsquigarrow$ (Halo2)
$s = \hat{s}$, $s_r < s_{r+1}$, $\prod_j (\gamma - K_j) = \prod_r (\gamma - s_r)$, $K_j = x_j 2^b + j$, $b = \lceil \log_2 n \rceil$,
$d = n \max(1, w - 1)$ for $w - 1$ carried columns

SORTED-INPUT $\{x$ non-decreasing$\}$ $(s, p) = sort(x) \rightsquigarrow$
$s_r = x_r$, $p_r = r$

SORTED-COUNTS $\{Pre_C\}$ $u = \#\{j : x_j \le t\} \rightsquigarrow$
s by one of the six constructions C above, $u = \hat{u}$, $0 \le u \le n$, $u > 0 \Rightarrow s_{u-1} \le t$, $u < n \Rightarrow t < s_u$

SEARCH-COUNT $\{Y$ non-decreasing$\}$ $u = search(Y, t) \rightsquigarrow$
$u = n - \sum_k [t < Y_k]$ (side right), $u = \sum_k [Y_k < t]$ (side left)

SEARCH-HINT $\{Y$ non-decreasing$\}$ $u = search(Y, t) \rightsquigarrow$
$u = \hat{u}$, $0 \le u \le n$, $u > 0 \Rightarrow Y_{u-1} \le t$, $u < n \Rightarrow t < Y_u$ (side right)

ARG-EXTREME $\{\top\}$ $u = argmax(x) \rightsquigarrow$
$c_j = [x_j > b_{j-1}]$, $b_j = \texttt{select}(c_j, x_j, b_{j-1})$, $u_j = \texttt{select}(c_j, j, u_{j-1})$, $b_0 = x_0$, $u_0 = 0$

ARG-EXTREME-SORTED $\{x$ non-decreasing$\}$ $u = argmax(x) \rightsquigarrow$
$u = \sum_{j<n-1} [x_j < x_{n-1}]$, or $u = n - 1$ if x increases strictly, $argmin(x) = 0$

ARG-EXTREME-SORTED-HINT $\{x$ non-decreasing$\}$ $u = argmax(x) \rightsquigarrow$
$u = \hat{u}$, $0 \le u < n$, $x_u = x_{n-1}$, $u = 0 \vee x_{u-1} < x_{n-1}$

(b) Scans, windows and scatters

A scan finds the first or last set flag of a vector $h \in \{0,1\}^n$, as `min` and `max` of `np.flatnonzero` do, or `np.argmax` of flags. A window is a slice $x[s:e]$ with runtime bounds and step 1 on up to three axes.

RUNNING-PRODUCTS $\{h \in \{0,1\}^n\}\ \ r = \mathit{first}(h) \rightsquigarrow$
$L_k = L_{k-1}(1-h_k),\ \ L_{-1} = 1,\ \ r = \sum_{k<n-1} L_k,\ \ e_k = L_{k-1} - L_k$
$\{r = \mathit{first}(h) \wedge e = E(r) \wedge T(r)\},\ \ \bigvee_k h_k$ asserted in the IR

SELECT-SCAN $\{h \in \{0,1\}^n\}\ \ r = \mathit{first}(h) \rightsquigarrow$
$r_k = \texttt{select}(h_k, k, r_{k+1})$ from $k = n-1$ down to 0, $\ r = r_0$

MONOTONE-SCAN $\{h$ non-decreasing$\}\ \ r = \mathit{first}(h) \rightsquigarrow$
$r = n - \sum_k h_k$, likewise for the other orders and for $h_k = [x_k \bowtie t]$ with x non-decreasing

MONOTONE-SCAN-HINT $\{x$ non-decreasing$\}\ \ r = \mathit{first}(([x_k \bowtie t])_k) \rightsquigarrow$
r is the position of t in x, checked as in SEARCH-HINT

SCATTER-ADD $\{\mathit{perm}(p)\}\ \ c[p] = v \rightsquigarrow$
$c_j = \sum_k [p_k = j]\, v_k \quad \{c = \mathit{scatter}(p, v)\},\ \ \{([p_k = j])_k = E(c_j)\}$ if $v_k = k + \text{const}$

SCATTER-WRITES $\{\mathit{perm}(p)\}\ \ c[p] = v \rightsquigarrow$
$c^0 = 0,\ \ c^{k+1} = c^k[p_k \mapsto v_k],\ \ c_j = c^n[j]$, lowered in the next round

SHIFT $\{[e = E(s)],\ [T(e)]\}\ \ z = x[s:e] \rightsquigarrow$
$z_u = \sum_d [s = d]\, [d + u < e]\, x_{u+d}$ on each axis

READ-OF-WRITE $\{\top\}\ \ y = x[i] \rightsquigarrow$
$y = v$ where every cell of x is $\texttt{select}([i = k], v, \cdot)$

ONE-HOT $\{e = E(r)\}\ \ y = x[r + c] \rightsquigarrow$
$y = \sum_k e_{k-c}\, x_k$

SELECT-CHAIN $\{\top\}\ \ y = x[i] \rightsquigarrow$
$y = \texttt{select}([i = n-1], x_{n-1}, \ldots \texttt{select}([i = 1], x_1, x_0))$

INDICATORS $\{\top\}\ \ y = x[i] \rightsquigarrow$
$e_k = [i = k],\ \ y = \sum_k e_k x_k,\ \ \sum_k e_k = 1 \quad \{y = x[i] \wedge 0 \le i < n\}$

(c) Insertions and masked writes

An insertion $a' = \mathit{insert}(a, x, e)$ inserts the row e into the n rows of a, each of w cells, at a runtime position x.

INSERT-SHARED $\{\top\}\ \ a' = \mathit{insert}(a, x, e) \rightsquigarrow$
$h = \dot{h} \in \{0,1\}^{n+1},\ \ \sum_k h_k = 1,\ \ P = \sum_k k\, h_k,\ \ (x - P)(x - P + n(1 - h_n)) = 0,\ \ g_k = \sum_{q<k} h_q,$
$t_k = g_{k+1}\, a_k,\ \ a'_k = a_k - t_k + t_{k-1} + h_k\, e \quad \{a' = \mathit{insert}(a, x, e) \wedge -n \le x \le n\}$

INSERT-COMPLEMENT $\{\top\}\ \ a' = \mathit{insert}(a, x, e) \rightsquigarrow$
h, P, g as in INSERT-SHARED, $\ q_k = \sum_{q \ge k} h_q,\ \ a'_k = q_{k+1}\, a_k + g_k\, a_{k-1} + h_k\, e \quad \{a' = \mathit{insert}(a, x, e) \wedge -n \le x \le n\}$

MASKED-WRITE $\{\top\}\ \ a' = a[m \mapsto v] \rightsquigarrow$
$a'_j = \texttt{select}(m_j, v_j, a_j)$

MASKED-WRITE-CASES $\{\top\}\ \ a' = a[m^1 \mapsto v^1] \cdots [m^q \mapsto v^q] \rightsquigarrow$
$a'_j = \sum_{C : \lambda_C \neq 0} [C]\, \lambda_C$ over the cases C of the comparisons in the masks

In MASKED-WRITE-CASES, λ_C is the value that a chain of masked writes leaves in the cell in case C.

(d) Element reads and arrays written at runtime

A wrap computes the position w that NumPy reads for an index $i \in [-n, n)$. A version tree $\overline{a}$ is an array together with all its writes $c' = c[i \mapsto v]$ at runtime positions under path conditions p, and the reads of the versions they produce. It has W writes, P of them under a path condition that is not `true`, and R reads.

Index-Bits $\{\top\}\ b = bits(i) \rightsquigarrow$
$i = \sum_l 2^l b_l,\ b_l(1-b_l) = 0,\ i < n,\ D = \lceil \log_2 n \rceil \quad \{0 \le i < n\}$

Bit-Mux $\{b = bits(i)\}\ y = x[i] \rightsquigarrow$
$m^0 = x,\ m_k^{l+1} = b_l\, m_{2k+1}^l + (1-b_l)\, m_{2k}^l,\ y = m_0^D$ for $n \ge 3$
$y = \texttt{select}(i, x_1, x_0),\ i(1-i) = 0 \quad \{0 \le i < 2\}$ for $n = 2$, without a precondition

Table $\{\top\}\ y = x[i] \rightsquigarrow$
$(i, y) \in \{(k, x_k) \mid k < n\}$ with $y = \hat{y}$ on Halo2 and a read-only array on Noir
$\{y = x[i] \wedge 0 \le i < n\}$

Wrap-Sign $\{\top\}\ w = wrap(i, n) \rightsquigarrow$
$s = \hat{s},\ s(1-s) = 0,\ w = \texttt{select}(s, i+n, i) \quad \{e = E(w)$ when used$\}$

Wrap-Hits $\{\top\}\ w = wrap(i, n) \rightsquigarrow$
$h = \hat{h} \in \{0,1\}^n,\ \sum_k h_k = 1,\ w = \sum_k k\, h_k,\ (i-w)(i-w+n) = 0$
$\{0 \le w < n \wedge -n \le i < n \wedge h = E(w)\}$

Wrap-Nonneg $\{i \ge 0\}\ w = wrap(i, n) \rightsquigarrow$
$w = i \quad \{e = E(w)$ when used$\}$

Select-Write $\{0 \le i < n,\ [e = E(i)]\}\ c' = c[i \mapsto v] \rightsquigarrow$
$c'_j = \texttt{select}([i = j] \wedge p, v, c_j)$ for every cell j that i can reach

Zero-Writes $\{0 \le i < n\}\ c' = c[i \mapsto 0] \rightsquigarrow$
$c'_j = c_j\,(1 - [i = j])$ for one-dimensional writes of 0 on every path

RAM-Trace $\{\tau \wedge chain(\overline{a})\}\ \overline{a} \rightsquigarrow$ (Halo2)
$S = \hat{S},\ S \equiv \overline{a},\ S$ sorted by $(i, t),\ i_r = i_{r-1} \wedge \text{read}_r \Rightarrow v_r = v_{r-1} \quad \{\overline{a}\}$

Native-RAM $\{chain(\overline{a})\}\ \overline{a} \rightsquigarrow$ (Noir)
a mutable array of Noir, every index checked by its RAM $\quad \{\overline{a}\}$

$chain(\overline{a})$ holds when the versions of $\overline{a}$ form one chain and each read reads a version whose cell no later write changes.

(e) Linear algebra

The operands are reals in fixed point, and `rescale` rounds a product sum once. A correlation $c = correlate(x, y)$ has $|x| + |y| - 1$ raw sums, and a matrix product $M = AB$ has A of size $m \times k$ and B of size $k \times n'$.

Product-Sums $\{\top\}\ c = correlate(x, y),\ M = AB \rightsquigarrow$
each used output is $\texttt{rescale}(\sum_t a_t b_t)$ over its T terms

Poly-Identity $\{\gamma\}\ c = correlate(x, y) \rightsquigarrow$ (Halo2)
$R = \hat{R},\ x(\gamma) \cdot y^{\text{rev}}(\gamma) = R(\gamma),\ c = \texttt{rescale}(R),\ d = |x| + |y| - 2$

Freivalds $\{\gamma\}\ M = AB \rightsquigarrow$ (Halo2)
$R = \hat{R},\ A(B\beta) = R\beta,\ M = \texttt{rescale}(R),\ \beta = (1, \gamma, \ldots, \gamma^{n'-1}),\ d = n' - 1$

$x(\gamma)$ is the polynomial with coefficients x, evaluated at γ.

(f) Statistics

Corr-Exact $\{\top\}\ r = corrcoef(x, y) \rightsquigarrow$
$A = nS_{xy} - S_xS_y,\ B = nS_{xx} - S_x^2,\ C = nS_{yy} - S_y^2$ exactly, $r = A/\sqrt{BC}$
for integers or Booleans and $n < 2^{31}$

Corr-Fixed $\{\top\}\ r = corrcoef(x, y) \rightsquigarrow$
means, deviations and three inner products in fixed point, then a root and a division

Here $S_x = \sum_k x_k$ and $S_{xy} = \sum_k x_k y_k$.

(g) Permutation checks

These check `is_permutation(p)` in a precondition of the program.

Perm-Hits $\{\top\}$ assert $perm(p) \rightsquigarrow$
$\sum_k [p_k = j] = 1$ for every $j < n$ $\{perm(p)\}$

Perm-Product $\{\gamma\}$ assert $perm(p) \rightsquigarrow$ (Halo2)
$\prod_k (\gamma - p_k) = \prod_k (\gamma - k),\ d = n$ $\{perm(p)\}$

B Correctness of Compilation

This appendix proves Theorem 1. The theorem assumes that the IR, read as a circuit, is correct, that the facts in the graph hold in every run that does not fail, and that every construction is correct, so the proof concerns only the selection conditions. We state the definitions of Sec. 5.4 precisely, together with the assumptions under which the theorem holds (Sec. B.1), and then prove one round of lowering, from which the theorem and lowering in several rounds follow (Sec. B.2).

B.1 Definitions and Assumptions

Assignments and phases. A circuit is a finite set of constraints over variables with values in $\mathbb{F}_p$, among which are the inputs u and the outputs y. Lookup and shuffle arguments, as in Table and RAM-Trace, count as constraints, which the proof system enforces up to its own cryptographic error. An *assignment* α gives every variable a value, and $\alpha \models C$ when α satisfies every constraint of C. The variables are split into a *first* and a *second phase*: the prover fixes the values α_1 of the first phase, the verifier then draws the challenge γ uniformly from $\mathbb{F}_p$, and the prover then chooses the values α_2 of the second phase. We write $(\alpha_1, \gamma, \alpha_2)$ for the assignment and $\Pr_\gamma$ for the probability over γ. A circuit that does not use the challenge mentions neither γ nor second-phase variables. Probabilities refer to this interactive protocol, which the proof system makes non-interactive with the Fiat–Shamir transform.

Definition 1 (Correctness up to an error). *A circuit C is* complete *if for every input u with $[\![P]\!](u) \neq \bot$ there are a first phase η_u with the inputs u and the outputs $[\![P]\!](u)$, and for every γ a second phase η_u^γ, such that $(\eta_u, \gamma, \eta_u^\gamma) \models C$. These assignments are* honest. *C is* sound up to ϵ *if for every input u, every output $y \neq [\![P]\!](u)$ and every first phase α_1 with the inputs u and the outputs y, $\Pr_\gamma[\exists \alpha_2.\ (\alpha_1, \gamma, \alpha_2) \models C] \leq \epsilon$.*

When $[\![P]\!](u) = \bot$, every output y differs from $[\![P]\!](u)$, so soundness bounds the acceptance of every output. Without a challenge, the probability is 0 or 1, and a circuit is complete and sound up to 0 exactly when it is correct in the sense of Sec. 3.1. The bound holds for each α_1 separately, so it also holds for a prover that picks the first phase at random.

The IR, requirements and nodes. Let C be the IR of the current round read as a circuit. For every output element e of an operator instance $f(a)$ lowered in this round, C contains the constraint $e = f(a)_e$, which no assignment satisfies where f is undefined on a, and for every assertion $\text{assert}_g\ \varphi$ it contains $g \Rightarrow \varphi$. The *retained* constraints $Ret(C)$ are those that mention γ or a second-phase variable. They are kept verbatim and generate no requirement, so operator constraints and assertions are those outside $Ret(C)$. The IR of symbolic execution has no retained constraint. $Std(C)$ contains the constraints that are neither operator constraints, assertions nor retained. A fact (φ, g) holds under α when $g \Rightarrow \varphi$ is true under α, and a value requirement, a variable x_r with a specification $spec_r$, holds when $spec_r(\alpha)$ is defined and equals $\alpha(x_r)$. The challenge γ and the RAM trace τ are requirements that hold under every assignment once a node provides them. The roots are the output elements, with the specifications $f(a)_e$, and the claims of the assertions.

A node n has constraints C_n, preconditions $pre(n)$ and postconditions $post(n)$. Its constraints mention variables of C, of value requirements and *private* variables of n. A node of a construction

is either *exact*, using no challenge, or *randomized*, with a check at γ of degree d_n. As in Sec. 5.2, *n provides r* if $r \in post(n)$, or if r is a fact implied by a postcondition of n, so r holds wherever $post(n)$ does. The graph also has *built-in* nodes: *standard nodes*, which cover $Std(C)$; the *retained node*, with the constraints $Ret(C)$ and no precondition or postcondition; *assertion nodes*, whose constraint $g \Rightarrow \varphi$ is also their postcondition; *fallback nodes* of the same form, for preconditions that no other node provides; and the node that provides γ.

Selections. A selection S with ranks $\rho(n) \in \mathbb{N}$ meets the conditions of Sec. 5.4: (S1) a node of S provides every root; (S2) for every $n \in S$ and $r \in pre(n)$, some $m \in S$ with $\rho(m) < \rho(n)$ provides r; and (S3) S contains every standard node and the retained node. The circuit obtained with S is $C' = \bigcup_{n \in S} C_n$, with the inputs and outputs of C. Let S_γ be the randomized nodes of S and $\epsilon(S) = \sum_{n \in S_\gamma} d_n / |\mathbb{F}_p|$.

Assumptions. Besides the hypotheses of Theorem 1, we assume the following, which the compiler is built to meet.

(A1) *Phases.* The inputs, the outputs, the variables of C other than those of $Ret(C)$, and every variable that occurs in a requirement other than γ and τ, including its specification and its guard, are in the first phase. Whether a requirement holds thus depends only on α_1, and we write $\alpha_1 \models r$.

(A2) *Shared values.* We fix one family of honest assignments η_u, η_u^γ of C, to which all statements below refer. A value requirement outside C, such as the bits of an index, has a total specification over the variables of C. For an honest η_u, the *common honest assignment* $\bar{\eta}_u$ extends η_u by $spec_r(\eta_u)$ for every such requirement r.

(A3) *Private variables.* The private variables of different nodes are distinct, and neither C nor any requirement mentions them.

(A4) *Built-in nodes.* Built-in nodes are exact, sound and complete in the sense of Def. 2. For fallback nodes, this means that their facts hold under $\bar{\eta}_u$.

(A5) *Small degrees.* $d_n < |\mathbb{F}_p|$ for every randomized node.

Definition 2 (Sound and complete nodes). *(a) An exact node n is* sound *if every $\alpha \models C_n$ with $\alpha \models pre(n)$ has $\alpha \models post(n)$. (b) A randomized node n is* sound *if every first phase α_1 with $\alpha_1 \models pre(n)$ and $\alpha_1 \not\models post(n)$ has $\Pr_\gamma[\exists \alpha_2.\ (\alpha_1, \gamma, \alpha_2) \models C_n] \le d_n / |\mathbb{F}_p|$. (c) A node n is* complete *if for every u with $[\![P]\!](u) \neq \bot$ and $\bar{\eta}_u \models pre(n)$, there are values θ_n of its private first-phase variables, and for every γ values θ_n^γ of its private second-phase variables, such that $(\bar{\eta}_u[\theta_n], \gamma, \eta_u^\gamma[\theta_n^\gamma]) \models C_n$, where $\alpha[\theta]$ is α updated with θ.*

Proving a randomized construction correct thus includes proving its degree bound, usually by the Schwartz–Zippel lemma [75].

B.2 The Proof

Lemma 1 (One round). *Suppose that (A1)–(A5) hold, that C is complete and sound up to ϵ, and that every node of S is sound and complete. Then C' is complete, with honest assignments that extend $\bar{\eta}_u$ and η_u^γ, and sound up to $\epsilon + \epsilon(S)$.*

Proof. *Completeness.* Fix u with $[\![P]\!](u) \neq \bot$. By induction on the rank, every $n \in S$ has $\bar{\eta}_u \models post(n)$ and private values $\theta_n, \theta_n^\gamma$ as in Def. 2(c). By (S2), the preconditions of n are provided by nodes of lower rank, so they hold under $\bar{\eta}_u$, and since n is complete, $\alpha^\gamma = (\bar{\eta}_u[\theta_n], \gamma, \eta_u^\gamma[\theta_n^\gamma]) \models C_n$ for every γ. Requirements mention no private and no second-phase variable, by (A1) and (A3), so α^γ satisfies the requirements that $\bar{\eta}_u$ satisfies. If n is exact, $\bar{\eta}_u \models post(n)$ by its soundness. If n is randomized, C_n is satisfiable for every γ after the first phase $\bar{\eta}_u[\theta_n]$, which by its soundness and

(A5) is possible only if $\bar{\eta}_u \models post(n)$. By (A3), the private values of all nodes combine into one assignment $(\eta'_u, \gamma, \eta'^{\gamma}_u)$ that agrees with each α^γ on the variables of its node, so it satisfies C' for every γ, with the inputs u and the outputs $[\![P]\!](u)$.

Soundness. Fix u, $y \neq [\![P]\!](u)$ and a first phase α_1 with the inputs u and the outputs y. By (A1), the set B of the nodes $n \in S_\gamma$ with $\alpha_1 \models pre(n)$ and $\alpha_1 \not\models post(n)$ is fixed before γ is drawn. Let $\alpha = (\alpha_1, \gamma, \alpha_2) \models C'$. If the postconditions of some node of S fail under α, take one, n, of least rank. By (S2), its preconditions hold, so n is not exact, since exact nodes are sound, and thus $n \in B$. Otherwise, all postconditions hold, so every root holds by (S1), and with $Std(C) \cup Ret(C) \subseteq C'$ by (S3), $\alpha \models C$. Hence C' is satisfiable only if C is or some C_n with $n \in B$ is, and the union bound gives the probability $\epsilon + \sum_{n \in B} d_n / |\mathbb{F}_p| \leq \epsilon + \epsilon(S)$. □

Proof of Theorem 1. The IR C of symbolic execution has no challenge. It is correct and satisfied by the values of every run that does not fail, so it is complete and sound up to 0. By Lemma 1, C' is complete and sound up to $\epsilon(S)$. If $S_\gamma = \emptyset$, C' uses no challenge and $\epsilon(S) = 0$, so C' is correct. Otherwise, for every input u, every output $y \neq [\![P]\!](u)$ and every first phase with them, C' is satisfiable with probability at most $\epsilon(S)$, the sum of $d/|\mathbb{F}_p|$ over the checks of S. □

The ranks order the justification of the requirements: the shared honest values are fixed by the run and their specifications, and the private values of each node are constructed relative to them.

Lowering in rounds. Let $C_0 = C$, and let C_{k+1} be obtained from C_k with a selection S_k. The operator instances that a construction introduces, such as the reads $x[p_r]$ of Sort-Hint, are lowered by later rounds. The checks and second-phase constraints of S_k are retained in every later round, the private first-phase variables of S_k become first-phase variables of C_{k+1}, and all rounds share one challenge.

Corollary 1 (Rounds). *If every round meets the assumptions of Lemma 1, then C_K is complete and sound up to $\sum_{k<K} \epsilon(S_k)$, and correct if no S_k contains a randomized node.*

Proof. By induction on K, with Lemma 1 in every round, which allows C_k to use the challenge. The bounds of the rounds add up by the union bound. □

Inputs outside the domain and failing runs are covered by soundness. An assignment that satisfies C_K satisfies C_0, up to the error, by the soundness argument of every round, and the assumed correctness of C_0 rejects these inputs.

C User Study Details

This appendix describes the web interface, the tasks and the assignment of tasks to participants in the user study of Sec. 8.3, and it reports the measurements of every task.

C.1 Web Interface

Participants took part through a personal link to a web interface that we built for the study. After an untimed practice exercise in each language, they worked on the six timed tasks one after another. Figure 12 shows the page of a timed task. Each run compiles the program and executes it on the inputs: a Zinnia program is executed as the compiled circuit and a Rust program in the RISC0 zkVM executor, in both cases without generating a proof. The timer counts down from 10 minutes and stops while the participant pauses, closes the page or fills in the submission form. When the time runs out, the last saved version is graded. The manual covers array operations, such as shapes, reductions, slicing, broadcasting, allocation and clamping, with examples in the language of the task.

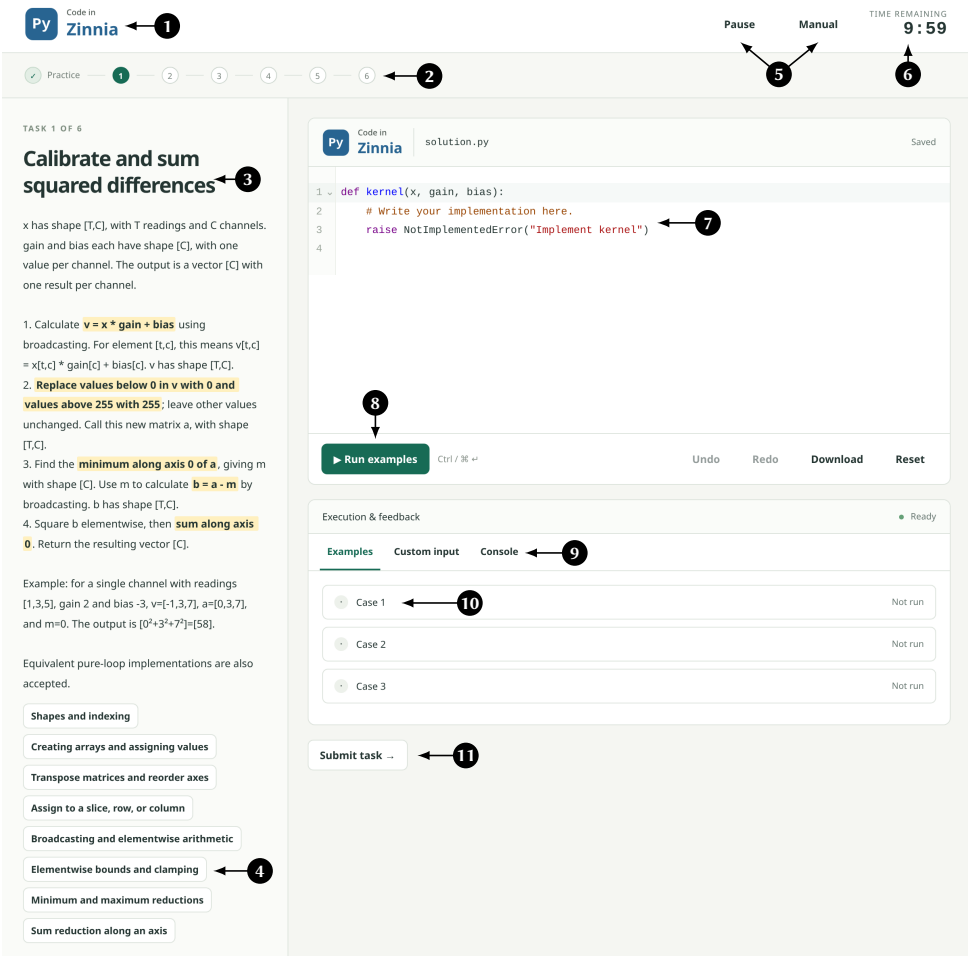


Fig. 12. **Page of a timed task in the study interface.** (1) Language of the task. (2) Progress through the practice and the six tasks. (3) Task description, with its main requirements highlighted. (4) Manual sections related to the task. (5) Pause button and the searchable manual. (6) Remaining time. (7) Editor with the starter code of the task. (8) Runs the visible examples. (9) Tabs for the visible examples, a custom input and the console with compiler and runtime messages. (10) Visible examples, each with its input, expected output and actual output. (11) Submits the task. The screenshot is taken from a preview session. The session label and the task names in the progress bar are hidden.

C.2 Tasks

Each task asks for the body of a function `kernel` over arrays of integers. Its description gives the input shapes, the steps of the computation and a worked example. With Zinnia, participants write the function in Python with the NumPy API, and with RISC0 in Rust with the ndarray API. Each task has visible examples and hidden tests.

T1 Channel energy. Inputs are a matrix `x` of shape $[T, C]$ and vectors `gain` and `bias` of shape $[C]$. The function computes `x` · `gain` + `bias` by broadcasting, clamps the result to $[0, 255]$, subtracts

the minimum of each column, and returns the sum of squares of each column, a vector of shape $[C]$.

T2 Cross products. Inputs are matrices `x` of shape $[T, C]$ and `y` of shape $[T, D]$. The function centers the columns as $z = T \cdot \mathtt{x} - s_x$ and $w = T \cdot \mathtt{y} - s_y$, where s_x and s_y are the column sums, and returns the matrix product $z^\top w$ of shape $[C, D]$.

T3 Weighted distances. Inputs are points `x` of shape $[N, 3]$ and `y` of shape $[M, 3]$ and a vector `weight` of three nonnegative weights. The function returns the matrix of shape $[N, M]$ whose entry (i, j) is the weighted squared distance $\sum_k \mathtt{weight}_k(\mathtt{x}_{ik} - \mathtt{y}_{jk})^2$.

T4 Image filter. Inputs are an image `x` of shape $[H, W]$ and a filter of shape $[U, V]$. The function returns the array of shape $[H - U + 1, W - V + 1]$ whose entry (i, j) is the sum of the elementwise products of the filter with the window of the image at row i and column j.

T5 Block energy. The input is an image `x` of shape $[H, W, C]$ with even H and W. For each channel of each non-overlapping 2×2 block, the function subtracts the smallest of the four values and sums the squares. The output has shape $[H/2, W/2, C]$.

T6 Masked diffusion. Inputs are a grid `x` of shape $[H, W]$ with values in $[0, 255]$ and a mask of 0s and 1s of the same shape. In each of three rounds, every interior cell whose mask is 1 becomes $\lfloor (4c + n_1 + n_2 + n_3 + n_4)/8 \rfloor$, where c is the cell and $n_1, \ldots, n_4$ are its neighbors in the grid of the previous round. The function returns the grid after the three rounds.

The practice exercise was the same in both languages. It subtracts a baseline from each column of a matrix, sets negative differences to zero and sums each row. The practice had no time limit and was not scored.

C.3 Assignment of Tasks

Each participant did every task once, three tasks with Zinnia and three with RISC0, in the order of Table 2. The orders come from a Williams design [93] for six tasks, which consists of six orders in which every task appears once at each position and every task directly follows every other task once. Each of these orders is used by two participants with opposite languages, which gives twelve orders. As a result, each task is done six times with each language and twice at each position, and every position has six tasks with each language. The twelve orders were assigned to the participant links at random. The language of the first practice exercise was also balanced. Each of its four combinations with the language of the first task occurs for three participants.

C.4 Detailed Results

Tables 3 and 4 report every task, and Table 5 summarizes each task with each language. A *test run* runs the program on the visible examples or on a custom input. The grading of the final version is not counted. Every test run is also graded on the hidden tests without showing the result to the participant. A *passing version* passes both the visible examples and the hidden tests. The *time to the first passing version* is the active time of the first test run with a passing version, or of the submission when the final version passes. It is 600 s when no version passes. The *active time* runs from the start of a task to its submission or to the time limit, without pauses, and is at most 600 s. A task *passes* when its final version passes all hidden tests. Every participant did three tasks with each language, so the averages over participants in Sec. 8.3 equal the averages over the 36 tasks of each language in the last row of Table 5.

Four tasks fail the hidden tests. With Zinnia, the program of P1 for T5 did not compile when the time ran out. With RISC0, the program of P0 for T5 did not compile when the time ran out, and P0 submitted T1 with wrong outputs after 15 test runs, the most of any task. P3 had a passing version of T2 after 223 s but changed it afterwards, and the version at the time limit gives wrong

Table 2. Task and language at each position of every participant's order (Z, Zinnia; R, RISC0), and the language of the first practice exercise.

	Position						First
Participant	1	2	3	4	5	6	practice
P0	T4 Z	T5 R	T3 R	T6 Z	T2 Z	T1 R	R
P1	T4 R	T5 Z	T3 Z	T6 R	T2 R	T1 Z	Z
P2	T1 R	T2 R	T6 Z	T3 R	T5 Z	T4 Z	R
P3	T2 R	T3 Z	T1 R	T4 Z	T6 Z	T5 R	Z
P4	T6 R	T1 Z	T5 Z	T2 R	T4 Z	T3 R	Z
P5	T6 Z	T1 R	T5 R	T2 Z	T4 R	T3 Z	R
P6	T3 R	T4 Z	T2 R	T5 R	T1 Z	T6 Z	R
P7	T5 R	T6 R	T4 Z	T1 Z	T3 R	T2 Z	R
P8	T5 Z	T6 Z	T4 R	T1 R	T3 Z	T2 R	Z
P9	T1 Z	T2 Z	T6 R	T3 Z	T5 R	T4 R	Z
P10	T3 Z	T4 R	T2 Z	T5 Z	T1 R	T6 R	Z
P11	T2 Z	T3 R	T1 Z	T4 R	T6 R	T5 Z	R

Table 3. Every task of participants P0 to P5, in the order performed. Lang.: Z, Zinnia; R, RISC0. Compile errors count the test runs that did not compile. First pass: active time of the first passing version (–, none). A task ends by submission or, marked †, at the 10-minute limit. Hidden tests: result of the final version.

Participant	Pos.	Task	Lang.	Test runs	Compile errors	First pass (s)	Active time (s)	Hidden tests
P0	1	T4 Image filter	Z	2	1	158	160	passed
	2	T5 Block energy	R	7	7	–	600†	failed, compile error
	3	T3 Weighted distances	R	8	3	395	398	passed
	4	T6 Masked diffusion	Z	3	0	272	274	passed
	5	T2 Cross products	Z	3	2	151	155	passed
	6	T1 Channel energy	R	15	8	–	594	failed, wrong output
P1	1	T4 Image filter	R	2	0	366	373	passed
	2	T5 Block energy	Z	3	3	–	600†	failed, compile error
	3	T3 Weighted distances	Z	2	0	164	169	passed
	4	T6 Masked diffusion	R	4	2	128	433	passed
	5	T2 Cross products	R	7	3	269	274	passed
	6	T1 Channel energy	Z	2	0	114	117	passed
P2	1	T1 Channel energy	R	2	1	497	522	passed
	2	T2 Cross products	R	1	0	142	153	passed
	3	T6 Masked diffusion	Z	2	1	446	451	passed
	4	T3 Weighted distances	R	4	3	394	400	passed
	5	T5 Block energy	Z	3	2	301	315	passed
	6	T4 Image filter	Z	3	2	230	234	passed
P3	1	T2 Cross products	R	4	0	223	600†	failed, wrong output
	2	T3 Weighted distances	Z	2	0	141	184	passed
	3	T1 Channel energy	R	3	0	155	600†	passed
	4	T4 Image filter	Z	2	0	70	104	passed
	5	T6 Masked diffusion	Z	1	0	56	508	passed
	6	T5 Block energy	R	3	0	411	457	passed
P4	1	T6 Masked diffusion	R	2	1	223	226	passed
	2	T1 Channel energy	Z	4	2	331	335	passed
	3	T5 Block energy	Z	1	0	198	203	passed
	4	T2 Cross products	R	1	0	167	171	passed
	5	T4 Image filter	Z	1	0	220	250	passed
	6	T3 Weighted distances	R	4	3	488	493	passed
P5	1	T6 Masked diffusion	Z	3	1	482	487	passed
	2	T1 Channel energy	R	1	0	205	212	passed
	3	T5 Block energy	R	6	5	559	563	passed
	4	T2 Cross products	Z	1	0	114	118	passed
	5	T4 Image filter	R	2	1	168	172	passed
	6	T3 Weighted distances	Z	1	0	103	108	passed

Table 4. Every task of participants P6 to P11, as in Table 3.

Participant	Pos.	Task	Lang.	Test runs	Compile errors	First pass (s)	Active time (s)	Hidden tests
P6	1	T3 Weighted distances	R	8	4	519	577	passed
	2	T4 Image filter	Z	4	0	372	449	passed
	3	T2 Cross products	R	6	2	488	521	passed
	4	T5 Block energy	R	4	2	546	553	passed
	5	T1 Channel energy	Z	2	1	240	245	passed
	6	T6 Masked diffusion	Z	8	5	583	599	passed
P7	1	T5 Block energy	R	1	0	514	564	passed
	2	T6 Masked diffusion	R	2	0	488	492	passed
	3	T4 Image filter	Z	3	2	350	388	passed
	4	T1 Channel energy	Z	1	0	336	359	passed
	5	T3 Weighted distances	R	2	0	382	399	passed
	6	T2 Cross products	Z	2	1	222	228	passed
P8	1	T5 Block energy	Z	1	0	109	118	passed
	2	T6 Masked diffusion	Z	1	0	92	98	passed
	3	T4 Image filter	R	4	3	559	563	passed
	4	T1 Channel energy	R	1	0	180	184	passed
	5	T3 Weighted distances	Z	1	0	213	218	passed
	6	T2 Cross products	R	7	6	572	575	passed
P9	1	T1 Channel energy	Z	3	0	293	297	passed
	2	T2 Cross products	Z	1	0	94	102	passed
	3	T6 Masked diffusion	R	10	8	519	522	passed
	4	T3 Weighted distances	Z	1	0	140	145	passed
	5	T5 Block energy	R	6	5	564	568	passed
	6	T4 Image filter	R	2	1	278	283	passed
P10	1	T3 Weighted distances	Z	1	0	134	137	passed
	2	T4 Image filter	R	1	0	244	249	passed
	3	T2 Cross products	Z	1	0	136	139	passed
	4	T5 Block energy	Z	1	0	194	197	passed
	5	T1 Channel energy	R	4	0	482	486	passed
	6	T6 Masked diffusion	R	2	0	417	444	passed
P11	1	T2 Cross products	Z	1	0	62	69	passed
	2	T3 Weighted distances	R	3	0	152	584	passed
	3	T1 Channel energy	Z	1	0	65	69	passed
	4	T4 Image filter	R	5	0	87	541	passed
	5	T6 Masked diffusion	R	1	0	72	600†	passed
	6	T5 Block energy	Z	1	0	45	92	passed

Table 5. Each task with each language, over the six participants who did it, and all tasks. Means of test runs, time to the first passing version (600 s when there is none) and active time, and the number of tasks whose final version passes the hidden tests.

	Test runs		First pass (s)		Active time (s)		Passed	
Task	Zinnia	RISC0	Zinnia	RISC0	Zinnia	RISC0	Zinnia	RISC0
T1 Channel energy	2.2	4.3	230	353	237	433	6/6	5/6
T2 Cross products	1.5	4.3	130	310	135	382	6/6	5/6
T3 Weighted distances	1.3	4.8	149	389	160	475	6/6	6/6
T4 Image filter	2.5	2.7	233	284	264	363	6/6	6/6
T5 Block energy	1.7	4.5	241	532	254	551	5/6	5/6
T6 Masked diffusion	3.0	3.5	322	308	403	453	6/6	6/6
All	2.0	4.0	218	363	242	443	35/36	33/36

outputs. Two other RISC0 tasks reached the time limit with a final version that passes, T1 of P3 and T6 of P11.

D Benchmark Programs

We list the programs of each corpus that the benchmark suite keeps, which make up the 327 programs of Sec. 8.1, and the programs that it filters out. DS-1000 programs are named by their problem number, NPBench programs by their benchmark, and Pythran programs by their test file, with the entry functions of files that have several.

DS-1000. Of the 220 NumPy problems, we keep 163: 291, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 309, 310, 311, 312, 313, 314, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 345, 346, 347, 348, 349, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 368, 369, 370, 373, 374, 375, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 398, 399, 400, 401, 402, 403, 405, 406, 407, 408, 409, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 422, 425, 426, 427, 428, 429, 430, 431, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 448, 449, 450, 451, 452, 453, 454, 456, 457, 458, 462, 463, 467, 468, 471, 472, 477, 478, 480, 481, 482, 483, 485, 489, 490, 491, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 507, 508, 509, 510. We filter out 57: 292, 293, 294, 305, 306, 307, 308, 315, 316, 333, 344, 350, 351, 352, 367, 371, 372, 376, 377, 378, 379, 380, 394, 395, 396, 397, 404, 410, 421, 423, 424, 432, 445, 446, 447, 455, 459, 460, 461, 464, 465, 466, 469, 470, 473, 474, 475, 476, 479, 484, 486, 487, 488, 492, 493, 494, 506.

NPBench. We keep all 54 benchmarks: `azimint_hist`, `azimint_naive`, `cavity_flow`, `channel_flow`, `compute`, `contour_integral`, `crc16`, `deep_learning/conv2d_bias`, `deep_learning/lenet`, `deep_learning/mlp`, `deep_learning/resnet`, `deep_learning/softmax`, `go_fast`, `mandelbrot1`, `mandelbrot2`, `nbody`, `polybench/adi`, `polybench/atax`, `polybench/bicg`, `polybench/cholesky`, `polybench/cholesky2`, `polybench/correlation`, `polybench/covariance`, `polybench/covariance2`, `polybench/deriche`, `polybench/doitgen`, `polybench/durbin`, `polybench/fdtd_2d`, `polybench/floyd_warshall`, `polybench/gemm`, `polybench/gemver`, `polybench/gesummv`, `polybench/gramschmidt`, `polybench/heat_3d`, `polybench/jacobi_1d`, `polybench/jacobi_2d`, `polybench/k2mm`, `polybench/k3mm`, `polybench/lu`, `polybench/ludcmp`, `polybench/mvt`, `polybench/nussinov`, `polybench/seidel_2d`, `polybench/symm`, `polybench/syr2k`, `polybench/syrk`, `polybench/trisolv`, `polybench/trmm`, `pythran/arc_distance`, `scattering_self_energies`, `spmv`, `stockham_fft`, `weather_stencils/hdiff`, `weather_stencils/vadv`.

Pythran. Of the 131 test files, 5 are variants of another file and are dropped: `_histogram`, `diffusion_pure_python`, `fbcorr`, `hyantes_core_numpy`, `pairwise`. Of the others, we filter out 11: `arc_distance_list`, `brownian`, `count_perm_sig`, `empirical`, `frequent_itemsets`, `grayscott`, `julia_pure_python`, `kmeans`, `loopy_jacob`, `monte_carlo_pricer`, `pi_buffon`. The remaining 115 files have 122 entry functions, of which we keep 110: `allpairs` (`sqr_dists`, `sqr_dists_loops`), `allpairs_distances`, `allpairs_distances_loops`, `another_quicksort`, `approximated_callgraph`, `arc_distance`, `babylonian`, `blacksholes`, `bubble_sort`, `calculate_u`, `caxpy`, `ccopy`, `cdotc`, `cdotu`, `check_mask`, `clip`, `clip2`, `collatz`, `collatz_modified`, `compute_subpix_2d_gaussian2`, `convnet`, `create_grid`, `cronbach`, `crotg`, `d2q9_nxnyns`, `deriv`, `deuxd_convolution`, `diffusion_numpy`, `emin`, `euclidean_distance_square`, `euler_challenge14`, `factorize_naive`, `fannkuch`, `fbcorr_numpy`, `fdtd`, `fft`, `fibo`, `fibo_seq`, `gauss`, `goodExpoMeasure`, `grouping`, `growcut`, `guerre`, `harris`, `hasting`, `histogram`, `hyantes_core`, `insertion_sort`, `ising`, `l2norm`, `lap`, `lapl2d`, `lapl3d`, `laplace`, `ldpc_decoder`, `ln`, `log_likelihood`, `lttb_pt`, `lu` (`factorMatrix0`, `factorMatrix1`), `make_circle_two_points`, `make_decision`, `mandel`, `matmul`, `matrix_class_distance`, `monte_carlo`, `morphology`, `mulmod`, `multi_export`, `multitype` (`check`, `check2`, `times`), `nd_local_maxima`, `nqueens`, `pairwise_numpy`, `periodic_bc`, `periodic_dist`, `perm`, `pivot`, `primes_sieve`, `primes_sieve2`, `projection_simplex`, `pselect`, `pythagorean_triples`, `queens_numba`, `quicksort`, `repeating`, `reverse_cumsum`, `rosen`, `sexy_primes`, `shallow_water`, `slowparts`, `smoothing`, `sobelfilter`, `specialconvolve`, `spectralDNS` (`add_pressure_diffusion`, `cross2`), `stone`, `sum_primes`, `sumarray3d`, `tax`, `train_eq` (`train_eq`, `train_eq2`), `value_index`, `vibr_energy`, `wave_simulation`, `wdist`, `weights`, `zero`. We filter out 12: `build_system`, `comp_unrolling`, `descent`, `euler_challenge13`, `extrema`, `ramsurf`, `rand_mat_stat`, `rc4`, `roman_decode`, `scrabble` (`scrabble_fun_score`, `scrabble_imp_score`), `train_equalizer`.

E Per-Program Evaluation Results

Tables 6, 7 and 8 give the per-program results behind Sec. 8.2 for the 214 programs that Zinnia accepts. Programs are named as in Appendix D and grouped by corpus. Programs marked † only copy data, and programs marked ‡ have outputs that follow from the input types alone. Sec. 8.2 leaves both groups out of its comparison of 172 programs.

Table 6 gives the size and the proving time of three circuits per backend: the compiled circuit (Zinnia), the hand-written circuit (Hand) and the direct translation (Direct). The size of a Halo2 circuit is its number of advice and lookup cells, and the size of a Noir circuit is its number of

UltraHonk gates. The proving time is in seconds and excludes setup. It is the median of three proofs with 8 threads, and the six circuits of a program are proved one after another in one session.

Table 7 first gives the proving time in seconds of each program on SP1, RISC0 and Cairo, and on SP1 and RISC0 with Groth16 wrapping. Each of these times is one proof. The core proofs use 8 threads and the wrapped proofs 16, and all zkVM proofs ran in sessions separate from those of Table 6. A dash marks a zkVM that was not run on the program. Cairo has no port of four of the programs marked †, and the wrapped proofs cover the 172 compared programs. The remaining columns give, for the 172 compared programs, the size of the compiled circuit with mechanisms of the compiler disabled, divided by its size with all of them enabled. Column All disables all the mechanisms that Sec. 8.2 lists, Fixed keeps one fixed construction per operator, Sel. disables the facts that selection uses, and Array disables the array facts. Column Mem. leaves runtime-indexed reads and writes to the backend's own memory. A ratio of exactly 1 is printed as 1.

Table 8 gives the verification time in milliseconds, the median of three verifications for each circuit and one verification for each zkVM. Verification is fast. Over all circuits, its median is 10.3 ms on Halo2 and 14.5 ms on Noir. Because of system noise, the range of the three verifications of one proof is 23.7% of their median for the median circuit. A difference of this size between the verification times of two circuits is therefore within the noise of the measurement.

Table 6. Circuit size and proving time of the compiled (Zinnia), hand-written (Hand) and directly translated (Direct) circuits.

	Halo2						Noir					
	Cells			Proving time (s)			Gates			Proving time (s)		
Program	Zinnia	Hand	Direct	Zinnia	Hand	Direct	Zinnia	Hand	Direct	Zinnia	Hand	Direct
DS-1000												
291‡	0	0	0	0.0201	0.0187	0.0201	34	34	34	0.0572	0.0543	0.0477
298	1,534	1,804	5,249	0.0863	0.0858	0.154	3,086	3,196	3,370	0.133	0.124	0.129
300	3,735	4,023	27,048	0.153	0.155	0.565	3,331	3,368	6,880	0.142	0.154	0.205
301†	940	1,040	1,040	0.0690	0.0709	0.103	2,941	2,941	2,941	0.118	0.137	0.112
302†	940	1,040	1,040	0.0963	0.0950	0.0860	2,941	2,941	2,941	0.170	0.127	0.170
303†	940	1,040	1,040	0.0853	0.100	0.0748	2,941	2,941	2,941	0.149	0.160	0.127
304†	940	1,040	1,040	0.0983	0.0818	0.0790	2,941	2,941	2,941	0.148	0.175	0.137
309	4,253	5,112	7,932	0.179	0.182	0.274	3,495	3,495	3,645	0.135	0.153	0.165
310	4,253	5,112	7,932	0.151	0.163	0.233	3,495	3,495	3,645	0.201	0.137	0.169
311	4,283	5,134	7,954	0.193	0.182	0.267	3,509	3,516	3,666	0.136	0.169	0.146
312	4,283	5,134	7,954	0.144	0.148	0.193	3,511	3,520	3,670	0.134	0.128	0.113
313	4,253	5,096	7,932	0.146	0.151	0.234	3,495	3,495	3,645	0.161	0.132	0.158
314	9,271	12,965	22,207	0.267	0.332	0.466	4,509	4,906	5,404	0.180	0.187	0.200
317†	35,000	35,000	35,000	0.580	0.524	0.489	9,066	9,066	9,066	0.265	0.321	0.292
318	3,066	3,330	10,639	0.128	0.155	0.195	3,440	3,466	3,519	0.122	0.121	0.130
319	14,832	24,264	95,529	0.339	0.422	1.10	7,066	7,786	7,867	0.188	0.177	0.193
320	2,663	3,782	4,930	0.130	0.146	0.157	3,250	3,259	3,351	0.143	0.129	0.130
321	2,663	3,782	4,930	0.120	0.108	0.142	3,250	3,259	3,351	0.104	0.112	0.110
322	527	629	1,586	0.0688	0.0596	0.0929	2,899	2,901	2,935	0.139	0.154	0.131
323	645	1,657	1,657	0.0613	0.0967	0.0955	2,904	2,960	2,960	0.136	0.121	0.119
324	649	1,661	1,661	0.0649	0.105	0.102	2,904	2,960	2,960	0.122	0.128	0.137
325	1,128	2,380	2,380	0.0840	0.118	0.124	2,983	3,104	3,104	0.148	0.121	0.131
326	1,372	1,564	1,568	0.0856	0.0868	0.0884	2,948	2,974	2,996	0.122	0.151	0.129
329	9,744	9,744	9,744	0.253	0.260	0.234	4,131	4,131	4,131	0.154	0.156	0.165
330	9,744	9,744	9,744	0.364	0.279	0.285	4,131	4,131	4,131	0.179	0.181	0.202
334	7,800	7,950	10,100	0.293	0.266	0.266	4,046	4,241	4,241	0.138	0.174	0.143
335	9,950	12,500	18,550	0.285	0.323	0.410	4,291	4,504	4,491	0.204	0.194	0.219
336†	416	416	416	0.0444	0.0463	0.0493	2,866	2,866	2,866	0.118	0.106	0.106
337†	416	416	416	0.0494	0.0477	0.0495	2,866	2,866	2,866	0.137	0.116	0.126
338†	832	832	832	0.0634	0.0773	0.0621	2,916	2,916	2,916	0.106	0.104	0.116
339†	832	832	832	0.0725	0.0684	0.0670	2,916	2,916	2,916	0.148	0.135	0.123

Table 6, continued.

	Halo2						Noir					
	Cells			Proving time (s)			Gates			Proving time (s)		
Program	Zinnia	Hand	Direct	Zinnia	Hand	Direct	Zinnia	Hand	Direct	Zinnia	Hand	Direct
340†	1,410	1,560	1,560	0.100	0.0826	0.0847	3,004	3,004	3,004	0.110	0.131	0.114
341†	1,410	1,560	1,560	0.0747	0.0860	0.0845	3,004	3,004	3,004	0.0944	0.0960	0.0962
342†	1,410	1,560	1,560	0.0977	0.0969	0.0889	3,004	3,004	3,004	0.115	0.111	0.128
343†	1,410	1,560	1,560	0.118	0.116	0.115	3,004	3,004	3,004	0.126	0.144	0.145
345	3,207	3,252	6,074	0.112	0.102	0.155	3,170	3,179	3,515	0.105	0.132	0.134
346	2,339	2,402	5,260	0.128	0.131	0.163	3,155	3,167	3,497	0.140	0.135	0.144
347	2,944	3,972	6,426	0.197	0.180	0.199	3,218	3,237	3,566	0.129	0.131	0.157
349‡	0	0	0	0.0159	0.0160	0.0209	34	34	34	0.0519	0.0476	0.0476
353	3,328	4,072	9,197	0.138	0.132	0.255	3,220	3,318	3,610	0.143	0.150	0.133
355	3,483	4,644	8,020	0.169	0.196	0.238	3,376	3,376	3,731	0.152	0.155	0.159
356	3,483	4,644	8,020	0.135	0.141	0.207	3,376	3,376	3,731	0.137	0.119	0.124
357	3,483	4,644	8,020	0.162	0.166	0.242	3,376	3,376	3,731	0.140	0.148	0.131
358	2,532	2,544	5,259	0.140	0.145	0.170	3,193	3,202	3,330	0.120	0.129	0.127
359†	360	360	360	0.0523	0.0518	0.0539	2,854	2,854	2,854	0.125	0.117	0.120
360†	624	720	720	0.0820	0.0849	0.0785	2,891	2,891	2,891	0.137	0.125	0.138
361†	1,248	1,248	1,248	0.0920	0.0891	0.0933	2,966	2,966	2,966	0.162	0.129	0.122
363	5,958	5,984	18,459	0.184	0.177	0.358	3,885	3,887	4,800	0.135	0.152	0.164
364	46,706	48,595	64,814	0.680	0.839	0.942	11,366	11,368	13,180	0.304	0.316	0.328
365	5,958	6,018	18,493	0.219	0.196	0.496	3,885	3,891	4,804	0.165	0.153	0.231
368	1,526	1,680	3,443	0.0990	0.111	0.115	3,003	3,025	3,024	0.124	0.0979	0.106
369	1,526	1,680	3,443	0.0959	0.114	0.134	3,002	3,025	3,024	0.159	0.160	0.149
370	1,526	1,680	3,443	0.0998	0.101	0.140	3,003	3,005	3,024	0.129	0.130	0.153
373‡	0	1,750	164,675	0.0190	0.112	2.32	34	3,029	28,125	0.0433	0.174	0.585
374	6,929	15,360	232,306	0.259	0.340	2.56	4,136	4,537	34,188	0.211	0.168	0.657
375	5,785	7,273	177,971	0.203	0.221	2.44	3,824	4,036	29,480	0.173	0.157	0.558
381	11,658	78,705	741,150	0.294	1.01	5.40	4,936	5,366	125,038	0.145	0.173	1.06
382	11,658	78,705	741,150	0.289	1.25	6.19	4,936	5,366	125,038	0.167	0.196	1.32
383	9,686	15,689	741,150	0.272	0.332	6.45	4,317	5,180	107,640	0.179	0.199	1.20
385†	700,000	700,000	700,000	6.16	6.32	6.33	127,816	127,816	127,816	1.35	1.31	1.27
386†	700,000	700,000	700,000	11.9	12.3	11.7	245,424	245,424	245,424	2.30	2.42	2.19
387†	700,000	700,000	700,000	6.49	6.24	6.36	127,816	127,816	127,816	1.59	1.33	1.39
389†	2,520	2,520	2,520	0.138	0.141	0.163	3,266	3,266	3,266	0.137	0.160	0.110
391†	2,100	2,100	2,100	0.115	0.137	0.135	3,191	3,191	3,191	0.136	0.153	0.110
392†	1,410	1,560	1,560	0.117	0.114	0.117	3,004	3,004	3,004	0.138	0.136	0.127
393†	2,100	2,100	2,100	0.120	0.123	0.111	3,191	3,191	3,191	0.109	0.106	0.119
398	1,711	1,871	2,016	0.0977	0.0935	0.115	3,070	3,070	3,070	0.120	0.118	0.128
399	1,842	2,007	2,292	0.108	0.116	0.135	3,102	3,102	3,102	0.134	0.143	0.152
400‡	0	0	0	0.0202	0.0231	0.0188	34	34	34	0.0513	0.0521	0.0472
401‡	0	0	0	0.0229	0.0214	0.0208	34	34	34	0.0583	0.0519	0.0549
402	38	43	386	0.0510	0.0306	0.0611	58	58	58	0.0811	0.0816	0.0775
403	38	43	46	0.0423	0.0316	0.0320	58	58	58	0.0748	0.0806	0.0736
409	3,552	3,552	3,552	0.136	0.134	0.129	3,320	3,320	3,320	0.134	0.134	0.132
412	1,146	1,770	4,341	0.0893	0.0934	0.136	3,078	3,079	3,369	0.126	0.129	0.149
413	1,818	1,972	4,361	0.0883	0.107	0.155	3,149	3,149	3,480	0.150	0.166	0.155
414	558	638	652	0.0615	0.0801	0.0704	2,875	2,875	2,875	0.120	0.124	0.114
415	770	952	1,490	0.0676	0.0715	0.0855	2,894	2,903	2,917	0.138	0.147	0.142
416	2,808	3,672	3,792	0.133	0.128	0.120	3,218	3,218	3,218	0.124	0.129	0.0999
417	558	638	652	0.0704	0.0805	0.0769	2,875	2,875	2,875	0.181	0.160	0.145
418	2,808	3,672	3,792	0.142	0.143	0.141	3,218	3,218	3,218	0.131	0.139	0.139
419	2,808	3,672	3,792	0.132	0.148	0.137	3,218	3,218	3,218	0.124	0.146	0.133
420	571	671	808	0.0758	0.0885	0.0736	2,867	2,880	2,877	0.152	0.134	0.181
422	7,428	13,650	26,150	0.253	0.343	0.528	6,149	6,149	6,149	0.259	0.223	0.216
428	1,963	2,247	3,418	0.114	0.119	0.129	3,069	3,094	3,086	0.135	0.128	0.148
429	1,963	2,247	3,418	0.107	0.111	0.121	3,069	3,094	3,086	0.126	0.126	0.138
430	1,963	2,247	3,418	0.120	0.130	0.126	3,069	3,086	3,086	0.144	0.116	0.153
431	1,681	2,019	3,934	0.0887	0.0999	0.131	3,028	3,053	3,101	0.120	0.111	0.113
433	1,048	1,220	1,469	0.0729	0.0675	0.0768	2,952	2,967	2,989	0.115	0.115	0.108
434	1,200	1,544	2,005	0.0778	0.0881	0.0868	2,972	3,009	3,077	0.129	0.130	0.138
435†	468	468	469	0.0555	0.0518	0.0524	2,872	2,872	2,872	0.137	0.132	0.121
436	4,180	4,560	7,670	0.178	0.154	0.224	3,531	3,574	3,766	0.141	0.141	0.123
437	4,180	4,560	7,670	0.192	0.146	0.195	3,531	3,574	3,766	0.134	0.135	0.141
438	898	934	3,374	0.0639	0.0931	0.153	2,904	2,930	3,057	0.112	0.156	0.0970
439	68,600	68,600	119,000	0.852	0.786	1.17	15,373	15,373	21,673	0.320	0.294	0.372
440	5,200	7,750	7,951	0.193	0.239	0.226	3,554	3,742	3,742	0.141	0.168	0.125
441	335	336	381	0.0559	0.0492	0.0552	2,848	2,848	2,853	0.120	0.121	0.107
442	3,866	3,893	12,082	0.125	0.128	0.277	3,669	3,687	5,219	0.138	0.121	0.193
443	3,786	3,813	12,002	0.121	0.143	0.257	3,654	3,690	5,219	0.137	0.133	0.228
444	6,306	6,610	20,186	0.180	0.190	0.353	4,029	4,030	5,479	0.131	0.136	0.181
448†	1,248	1,248	1,248	0.0732	0.0994	0.0833	2,966	2,966	2,966	0.120	0.138	0.111
449†	1,248	1,248	1,248	0.0797	0.0952	0.0971	2,966	2,966	2,966	0.124	0.129	0.131
450	17,744	21,994	50,246	0.453	0.419	0.769	6,196	6,673	7,294	0.202	0.177	0.192
451‡	0	0	0	0.0239	0.0231	0.0209	34	34	34	0.0564	0.0520	0.0641
452	3,860	4,020	5,881	0.155	0.159	0.174	3,342	3,347	3,607	0.127	0.126	0.127

Table 6, continued.

	Halo2						Noir					
	Cells			Proving time (s)			Gates			Proving time (s)		
Program	Zinnia	Hand	Direct	Zinnia	Hand	Direct	Zinnia	Hand	Direct	Zinnia	Hand	Direct
453	3,410	3,800	6,061	0.128	0.146	0.163	3,275	3,317	3,573	0.147	0.140	0.126
454	4,345	4,865	7,795	0.184	0.189	0.229	3,422	3,422	3,844	0.157	0.154	0.183
456	4,844	6,273	14,216	0.147	0.192	0.336	3,611	3,727	4,392	0.146	0.147	0.198
457	6,272	7,701	17,360	0.189	0.187	0.349	3,881	3,997	4,830	0.129	0.134	0.176
458	6,272	7,701	17,361	0.213	0.221	0.318	3,825	3,941	4,774	0.131	0.152	0.192
462	3,701	3,789	11,326	0.140	0.139	0.289	3,649	3,650	5,096	0.180	0.156	0.209
463	3,701	3,789	11,326	0.108	0.104	0.197	3,649	3,650	5,096	0.106	0.117	0.151
467†	832	832	832	0.0757	0.0643	0.0767	3,084	3,084	3,084	0.139	0.113	0.125
468†	832	832	832	0.0836	0.0742	0.0765	3,084	3,084	3,084	0.121	0.132	0.155
471†	79,380	79,380	79,380	1.02	0.946	0.952	16,991	16,991	16,991	0.394	0.355	0.381
472†	52,920	52,920	52,920	0.750	0.765	0.659	12,266	12,266	12,266	0.276	0.332	0.301
477†	240	240	241	0.0583	0.0522	0.0560	2,841	2,841	2,841	0.195	0.136	0.141
478	240	240	241	0.0583	0.0590	0.0611	2,841	2,841	2,841	0.157	0.161	0.151
480	1,204	1,225	2,688	0.0901	0.0869	0.132	2,963	2,963	3,134	0.138	0.129	0.158
481	1,652	1,774	3,173	0.0955	0.0990	0.106	3,098	3,103	3,274	0.140	0.118	0.146
485	9,045	9,045	10,306	0.205	0.202	0.203	4,234	4,234	4,234	0.177	0.187	0.143
490	34,980	37,980	1,997,601	0.691	0.765	13.7	7,546	8,089	346,761	0.219	0.235	3.77
491	1,660	2,296	2,605	0.0937	0.0968	0.105	3,024	3,115	3,077	0.121	0.142	0.141
500†	1,890	1,890	1,890	0.113	0.0924	0.101	3,154	3,154	3,154	0.111	0.122	0.105
501	14,027	26,364	26,364	0.285	0.453	0.444	4,929	6,661	7,759	0.158	0.177	0.188
502	14,027	26,364	26,364	0.329	0.514	0.513	4,929	6,661	7,759	0.195	0.226	0.214
503	29,913	57,291	57,291	0.642	0.907	0.924	8,990	11,013	10,931	0.253	0.270	0.275
504	30,418	57,796	57,968	0.561	0.768	0.733	8,822	10,845	10,651	0.256	0.291	0.258
505	32,953	60,333	61,352	0.683	0.928	0.915	8,935	10,845	10,932	0.236	0.287	0.296
507	2,963	3,194	33,710	0.121	0.116	0.484	3,352	3,468	4,808	0.108	0.115	0.146
508	9,288	10,456	618,014	0.234	0.223	4.40	4,988	5,750	65,221	0.157	0.188	0.777
509	9,691	10,470	209,797	0.225	0.270	2.22	5,063	5,617	8,752	0.167	0.173	0.298
510	9,609	10,462	210,597	0.182	0.244	2.24	4,954	5,749	8,752	0.163	0.167	0.308
NPBench												
compute	39,786	40,553	73,833	0.634	0.602	0.878	8,397	8,397	10,061	0.221	0.188	0.250
crc16	134,219	134,875	141,288	1.46	1.47	1.65	18,882	45,256	43,657	0.382	0.555	0.552
deep_learning/conv2d_bias	657,400	657,400	938,201	5.05	4.83	6.20	127,033	127,033	127,033	1.30	1.39	1.27
deep_learning/mlp	12,238	14,662	15,992	0.259	0.312	0.331	4,557	4,562	4,593	0.190	0.185	0.178
deep_learning/resnet	1,641,416	5,174,103	5,925,511	12.8	34.8	37.5	251,116	586,040	606,620	1.96	4.98	5.12
deep_learning/softmax	892,736	1,390,400	1,571,584	7.17	11.7	12.0	174,920	177,535	188,671	1.60	1.58	2.17
go_fast	35,630	50,942	51,505	0.596	0.665	0.796	10,004	10,004	10,004	0.227	0.230	0.225
polybench/atax	10,625	10,625	10,625	0.204	0.212	0.243	4,397	4,397	4,397	0.158	0.152	0.154
polybench/bicg	11,080	11,080	11,080	0.231	0.267	0.244	4,479	4,479	4,479	0.192	0.179	0.195
polybench/cholesky	27,341	29,100	39,145	0.474	0.495	0.618	7,036	7,079	8,265	0.175	0.179	0.237
polybench/covariance2	25,238	25,553	54,792	0.487	0.535	0.830	9,240	9,748	11,819	0.266	0.293	0.330
polybench/deriche	111,547	113,552	122,983	1.40	1.82	1.96	21,628	21,634	21,648	0.389	0.383	0.420
polybench/durbin	12,613	13,352	13,890	0.226	0.258	0.260	5,604	5,651	5,654	0.145	0.165	0.140
polybench/floyd_warshall	357,120	430,848	537,344	3.93	4.53	4.71	55,490	76,546	83,630	0.715	1.05	1.03
polybench/gemm	61,962	70,014	70,014	1.02	1.14	1.23	13,774	13,774	13,774	0.300	0.297	0.267
polybench/gemver	62,614	62,886	65,415	0.813	0.834	1.08	12,768	12,768	12,768	0.360	0.293	0.259
polybench/gesummv	50,518	50,518	50,518	0.960	0.975	0.930	9,396	9,396	9,396	0.263	0.328	0.290
polybench/gramschmidt	85,350	88,986	104,947	1.08	1.36	1.67	17,788	17,852	17,802	0.380	0.384	0.357
polybench/k2mm	83,714	99,267	99,267	1.26	1.30	1.61	19,252	19,252	19,252	0.388	0.344	0.416
polybench/k3mm	68,319	89,579	133,832	1.33	1.47	2.02	19,559	19,559	19,559	0.411	0.391	0.389
polybench/lu	33,880	35,637	35,753	0.497	0.519	0.552	8,871	9,402	9,402	0.193	0.194	0.196
polybench/ludcmp	38,298	40,323	42,091	0.612	0.574	0.666	9,346	9,962	9,962	0.254	0.238	0.301
polybench/mvt	14,560	14,560	14,560	0.276	0.270	0.257	4,951	4,951	4,951	0.150	0.191	0.156
polybench/symm	82,829	83,531	94,400	0.826	0.873	0.912	13,596	13,596	13,596	0.211	0.210	0.211
polybench/syr2k	129,108	129,108	209,279	1.52	1.53	2.47	19,170	19,170	19,170	0.437	0.369	0.394
polybench/syrk	87,605	87,605	88,550	1.73	1.76	1.80	14,032	14,032	14,032	0.315	0.313	0.382
polybench/trisolv	8,567	9,303	9,852	0.236	0.287	0.274	4,015	4,046	4,046	0.181	0.166	0.164
polybench/trmm	36,381	36,381	42,494	0.591	0.598	0.606	8,507	8,507	8,507	0.222	0.228	0.304
pythran/arc_distance	52,448	65,216	69,728	0.689	0.991	1.14	11,219	11,276	11,276	0.285	0.238	0.294
scattering_self_energies	10,276,644	10,412,892	24,184,692	88.3	77.8	170	1,711,468	1,712,048	1,712,099	12.1	12.1	12.3
weather_stencils/hdiff	519,940	539,300	661,981	5.22	5.15	6.63	75,429	75,429	75,429	1.20	0.969	1.01
weather_stencils/vadv	1,960,910	2,022,110	2,544,786	15.5	15.6	21.6	293,764	293,764	325,432	3.39	3.21	3.46
Pythran												
allpairs (sqr_dists)	275,456	275,456	280,065	2.78	2.58	3.08	39,559	39,559	39,559	0.632	0.647	0.667
allpairs (sqr_dists_loops)	275,456	275,456	280,065	2.52	2.63	2.69	39,559	39,559	39,559	0.551	0.533	0.526
arc_distance	16,390	20,380	21,790	0.408	0.404	0.413	5,448	5,483	5,483	0.163	0.177	0.165
bubble_sort	3,762	5,412	7,700	0.151	0.168	0.198	3,482	3,758	3,958	0.135	0.163	0.155
ccopy	189,968	443,354	1,109,257	2.01	3.47	7.29	79,496	82,318	120,714	1.09	1.01	1.17
clip	2,065	2,639	3,469	0.130	0.126	0.171	3,097	3,112	3,211	0.143	0.149	0.133
create_grid†	560	560	561	0.160	0.153	0.149	3,908	3,908	3,908	0.140	0.133	0.150
cronbach	15,012	15,048	17,071	0.295	0.394	0.381	5,078	5,248	5,253	0.206	0.181	0.199
crotg	3,794	3,812	4,787	0.146	0.159	0.159	3,287	3,296	3,338	0.128	0.130	0.138
d2q9_nxnyns	479,304	480,480	1,022,225	4.56	3.96	7.94	66,919	67,110	74,595	1.05	1.08	1.01
deuxd_convolution	47,323	49,423	386,136	0.675	0.653	3.62	9,125	9,125	9,125	0.240	0.214	0.196

Table 6, continued.

	Halo2						Noir					
	Cells			Proving time (s)			Gates			Proving time (s)		
Program	Zinnia	Hand	Direct	Zinnia	Hand	Direct	Zinnia	Hand	Direct	Zinnia	Hand	Direct
euclidean_distance_square	1,794	1,988	2,098	0.0949	0.110	0.0974	3,009	3,009	3,009	0.122	0.119	0.128
fbcorr_numpy	1,755,654	1,756,038	1,784,070	15.3	15.1	15.2	221,600	221,600	221,600	2.21	2.14	2.13
fibo_seq	59,211	70,790	107,735	0.690	0.818	1.01	14,622	17,400	17,400	0.267	0.277	0.272
goodExpoMeasure	796,724	1,160,800	1,215,523	6.30	9.13	10.6	150,018	152,122	155,513	1.56	1.65	1.59
harris	6,970	6,985	7,138	0.174	0.180	0.192	3,731	3,731	3,731	0.139	0.137	0.147
hasting	1,469	1,630	2,274	0.0702	0.0727	0.0808	2,967	2,967	2,967	0.0938	0.0968	0.0931
insertion_sort	3,809	4,988	45,188	0.152	0.152	0.711	3,646	3,677	8,390	0.133	0.140	0.268
ising	3,651	5,559	5,816	0.115	0.140	0.142	3,547	3,575	3,564	0.0991	0.102	0.0990
l2norm	26,096	26,512	39,104	0.458	0.455	0.593	6,019	6,096	7,696	0.170	0.158	0.176
lap	5,124	5,344	7,396	0.168	0.159	0.199	3,492	3,505	3,506	0.113	0.126	0.125
lap13d	139,442	140,606	248,109	1.31	1.35	2.15	26,478	27,654	34,844	0.371	0.352	0.487
ln	13,808	14,160	27,216	0.320	0.303	0.558	4,869	4,869	4,971	0.199	0.193	0.218
log_likelihood	35,103	44,296	43,891	0.622	0.753	0.752	9,001	9,008	8,934	0.333	0.261	0.266
lu (factorMatrix0)	6,263	6,351	12,822	0.207	0.196	0.268	3,949	4,450	5,451	0.125	0.190	0.189
lu (factorMatrix1)	6,155	6,217	11,911	0.192	0.182	0.271	3,894	4,410	5,262	0.150	0.172	0.222
make_decision	48,816	53,504	65,969	0.956	0.983	0.957	8,793	9,488	9,428	0.265	0.247	0.292
matmul	259,072	259,072	264,192	2.39	2.39	2.45	35,463	35,463	35,463	0.500	0.562	0.493
multi_export†	60	60	60	0.0347	0.0364	0.0390	2,822	2,822	2,822	0.124	0.142	0.139
multitype (check)	131	132	144	0.0492	0.0478	0.0493	2,831	2,831	2,831	0.157	0.159	0.141
multitype (check2)	73	73	77	0.0353	0.0407	0.0437	2,825	2,825	2,825	0.111	0.135	0.162
multitype (times)	54	60	178	0.0297	0.0326	0.0425	388	388	388	0.0548	0.0548	0.0573
pairwise_numpy	155,921	162,160	175,489	1.69	2.15	2.48	22,779	23,552	23,552	0.398	0.339	0.351
periodic_bc†	936	936	937	0.0592	0.0629	0.0608	2,947	2,947	2,947	0.107	0.114	0.107
periodic_dist	213,885	245,682	17,793,125	2.77	3.02	85.9	28,910	32,202	2,361,022	0.627	0.578	38.2
perm†	208	208	208	0.0491	0.0529	0.0512	3,025	3,025	3,025	0.162	0.141	0.135
projection_simplex	3,897	4,876	14,409	0.153	0.169	0.283	3,318	3,381	4,695	0.127	0.123	0.168
pythagorean_triples	259	264	397	0.0404	0.0416	0.0485	2,847	2,847	2,847	0.100	0.0945	0.0949
reverse_cumsum	548	636	641	0.0530	0.0723	0.0748	2,887	2,887	2,887	0.121	0.102	0.108
rosen	3,288	3,348	3,381	0.115	0.115	0.110	3,232	3,232	3,232	0.0947	0.0957	0.0941
smoothing	3,173	3,173	3,229	0.133	0.111	0.118	3,152	3,152	3,152	0.119	0.119	0.126
specialconvolve	12,460	12,488	13,888	0.322	0.269	0.299	4,716	4,716	4,716	0.206	0.171	0.178
spectralDNS (add_pressure_diffusion)	46,550	46,550	245,082	0.709	0.766	2.53	8,721	8,721	362,181	0.268	0.257	3.25
spectralDNS (cross2)	62,930	62,930	75,810	1.13	0.966	1.35	11,241	11,241	11,241	0.270	0.273	0.255
sumarray3d	169,168	180,432	206,177	2.03	2.21	2.32	23,587	24,444	27,388	0.414	0.526	0.405
tax	10,489	12,715	22,162	0.237	0.269	0.343	4,446	4,524	5,702	0.191	0.172	0.175
value_index	4,810	8,153	34,515	0.262	0.215	0.557	3,961	4,201	5,113	0.184	0.178	0.245
vibr_energy	15,376	23,056	23,056	0.362	0.453	0.437	5,751	5,751	5,751	0.177	0.180	0.181
wdist	5,025	5,523	6,904	0.187	0.184	0.189	3,438	3,535	3,497	0.142	0.165	0.140
weights	1,815	2,513	5,386	0.0932	0.121	0.177	3,042	3,057	3,264	0.142	0.154	0.167

Table 7. Proving time of the zkVMs, and circuit size of the compiled circuits with mechanisms of the compiler disabled.

	zkVM proving time (s)					Size with mechanisms disabled, over the size with all enabled									
	Core proof			Groth16		Halo2 cells					Noir gates				
Program	SP1	RISC0	Cairo	SP1	RISC0	All	Fixed	Sel.	Array	Mem.	All	Fixed	Sel.	Array	Mem.
DS-1000															
291‡	25.3	3.73	14.2	–	–										
298	23.2	3.70	16.0	325	53.9	1.96	1	1.94	1.02	1.03	1.09	1	1.11	1.01	1.27
300	22.5	3.73	15.8	334	57.1	1.69	1.69	1	1	1	1.27	1.27	1	1	1
301†	25.2	3.62	24.3	–	–										
302†	23.3	3.85	23.7	–	–										
303†	23.6	3.53	15.0	–	–										
304†	23.8	3.76	18.2	–	–										
309	23.7	3.60	16.4	331	64.2	1	1	1	1	1	1	1	1	1	1
310	24.0	3.67	14.4	336	50.6	1	1	1	1	1	1	1	1	1	1
311	24.2	3.64	14.6	354	56.7	1	1	1	1	1	1	1	1	1	1
312	22.6	3.69	15.2	333	59.0	1	1	1	1	1	1	1	1	1	1
313	23.3	4.12	21.8	334	57.5	1	1	1	1	1	1	1	1	1	1
314	24.7	3.84	14.4	352	53.6	1.06	1.06	1	1	1.20	1.04	1.04	1	1	1.05
317†	27.0	14.8	15.6	–	–										
318	23.6	3.72	13.9	359	72.9	1.45	1.41	1.07	1.22	1.06	1.09	1.09	1.01	1.01	1.00
319	25.6	13.7	17.8	330	64.6	2.23	2.20	1.02	1.03	1	1.51	1.50	1.00	1.01	1
320	24.9	3.36	14.9	326	55.5	1	1	1	1	1	1	1	1	1	1
321	21.1	3.15	12.5	342	53.4	1	1	1	1	1	1	1	1	1	1
322	22.2	3.36	12.8	332	59.8	1	1	1	1	1	1.00	1.00	1	1	1

Table 7, continued.

	zkVM proving time (s)					Size with mechanisms disabled, over the size with all enabled									
	Core proof			Groth16		Halo2 cells					Noir gates				
Program	SP1	RISC0	Cairo	SP1	RISC0	All	Fixed	Sel.	Array	Mem.	All	Fixed	Sel.	Array	Mem.
323	24.6	3.26	16.8	333	50.3	1	1	1	1	1	1	1	1	1	1
324	23.5	3.39	15.7	323	57.0	1	1	1	1	1	1	1	1	1	1
325	23.4	3.74	13.8	347	50.5	1	1	1	1	1	1	1	1	1	1
326	23.7	3.56	14.4	322	61.1	1.05	1	1.05	1.05	1	1.00	1	1.00	1.00	1
329	24.1	7.04	15.2	332	68.7	1	1	1	1	1	1	1	1	1	1
330	22.4	6.83	20.3	335	60.0	1	1	1	1	1	1	1	1	1	1
334	23.8	7.24	15.2	338	63.3	1	1	1	1	1	1	1	1	1	1
335	24.9	7.28	16.1	340	59.2	1	1	1	1	1	1.01	1.01	1	1	1
336†	23.1	3.63	14.3	–	–										
337†	24.3	3.68	16.6	–	–										
338†	22.8	3.44	16.0	–	–										
339†	24.0	3.40	13.8	–	–										
340†	25.5	4.19	16.2	–	–										
341†	23.7	3.65	17.3	–	–										
342†	25.2	3.77	14.8	–	–										
343†	24.1	3.63	16.6	–	–										
345	25.4	3.84	14.7	347	62.6	1.00	1.00	1.00	1	0.96	1.01	1.01	1.00	1	1.02
346	23.0	3.63	19.1	256	43.5	1.01	1.01	1.00	1	1.24	1.02	1.02	1.00	1	1.02
347	24.7	7.33	14.9	260	42.8	1.01	1.01	1.00	1	1.26	1.02	1.02	1.00	1	1.02
349‡	24.9	3.54	16.8	–	–										
353	23.5	3.52	16.4	328	57.4	1.28	1	1	1	1	1.07	1.01	1	1	1
355	22.8	3.39	15.2	332	61.6	1	1	1	1	1	1	1	1	1	1
356	26.1	4.00	36.6	337	58.2	1	1	1	1	1	1	1	1	1	1
357	23.6	3.93	38.9	339	56.2	1	1	1	1	1	1	1	1	1	1
358	24.5	3.90	18.5	341	65.5	1	1	1	1	1	1.02	1.02	1	1	1
359†	24.9	3.78	13.5	–	–										
360†	23.3	3.43	15.9	–	–										
361†	24.9	3.91	16.7	–	–										
363	21.3	6.85	12.3	335	60.0	1.01	1	1.01	1	1	1.00	1	1.00	1	1
364	24.7	14.1	15.0	318	68.7	1.00	1	1.00	1	1	1.00	1	1.00	1	1
365	24.2	7.28	13.7	340	62.1	1.01	1	1.01	1	1	1.00	1	1.00	1	1
368	22.8	3.46	14.5	339	54.3	1	1	1	1	1	1	1	1	1	1
369	24.9	3.67	16.2	381	67.8	1	1	1	1	1	1	1	1	1	1
370	22.8	3.55	13.9	346	62.8	1	1	1	1	1	1	1	1	1	1
373‡	24.0	26.3	13.0	–	–										
374	23.7	7.07	14.0	325	68.0	1.97	1.97	1	1	1	1.09	1.09	0.97	1	1
375	21.9	6.82	14.0	384	68.6	4.35	4.35	1	1	0.93	2.08	2.08	1	1	1
381	26.0	7.31	17.0	339	70.5	5.66	5.66	1.05	1	1	3.27	3.27	0.96	1	1
382	25.8	7.37	16.2	336	51.1	5.66	5.66	1.05	1	1	3.27	3.27	0.96	1	1
383	25.2	6.99	19.0	352	63.1	4.69	4.69	1	1	1	2.99	2.99	1.00	1	1
385†	74.9	221	17.4	–	–										
386†	329	760	15.6	–	–										
387†	76.6	226	15.1	–	–										
389†	22.4	6.82	15.8	–	–										
391†	23.1	7.22	13.8	–	–										
392†	23.3	3.49	14.8	–	–										
393†	21.5	6.81	13.8	–	–										
398	22.0	3.48	14.3	334	60.8	1	1	1	1	1	1	1	1	1	1
399	23.9	3.65	14.4	323	54.0	1	1	1	1	1	1	1	1	1	1
400‡	21.9	3.50	14.5	–	–										
401‡	21.2	3.48	12.4	–	–										
402	24.0	3.44	15.2	342	53.4	1	1	1	1	1	1	1	1	1	1
403	23.2	3.92	18.0	330	59.0	1	1	1	1	1	1	1	1	1	1
409	23.5	3.58	17.3	357	56.8	1	1	1	1	1	1	1	1	1	1
412	23.7	3.72	14.8	389	69.3	1	1	1	1	1	1	1	1	1	1
413	22.3	3.42	16.7	348	72.7	1	1	1	1	1	1	1	1	1	1
414	22.0	3.58	13.4	334	61.8	1	1	1	1	1	1	1	1	1	1
415	24.8	3.83	16.3	327	58.8	1	1	1	1	1	1.00	1.00	1	1	1
416	24.8	7.21	24.0	328	58.3	1	1	1	1	1	1	1	1	1	1
417	24.8	3.63	15.2	336	67.7	1	1	1	1	1	1	1	1	1	1
418	22.8	7.22	14.4	259	45.6	1	1	1	1	1	1	1	1	1	1
419	24.5	7.17	16.3	357	64.0	1	1	1	1	1	1	1	1	1	1
420	23.4	3.51	14.7	367	69.7	1	1	1	1	1	1	1	1	1	1
422	22.1	26.5	14.9	341	74.6	1.84	1.84	1	1	1	1	1	1	1	1
428	21.3	3.21	13.1	349	60.6	1	1	1	1	1	1	1	1	1	1
429	25.4	3.54	14.4	329	61.2	1	1	1	1	1	1	1	1	1	1
430	23.3	6.99	15.7	337	60.6	1	1	1	1	1	1	1	1	1	1
431	22.2	6.89	15.5	340	60.2	1	1	1	1	1	1	1	1	1	1
433	24.3	3.49	15.9	342	64.5	1.03	1.03	1.02	1	1.14	1.01	1.01	1.01	1	1.31
434	23.0	3.73	14.0	335	63.9	1.18	1.18	1.11	1	1.10	1.02	1.02	1.01	1	1.60
435†	22.3	3.54	15.1	–	–										
436	25.2	3.54	21.2	334	63.4	1	1	1	1	1	1.04	1.04	1	1	1
437	23.4	3.94	13.8	326	65.8	1	1	1	1	1	1.04	1.04	1	1	1

Table 7, continued.

	zkVM proving time (s)					Size with mechanisms disabled, over the size with all enabled									
	Core proof			Groth16		Halo2 cells					Noir gates				
Program	SP1	RISC0	Cairo	SP1	RISC0	All	Fixed	Sel.	Array	Mem.	All	Fixed	Sel.	Array	Mem.
438	25.1	7.27	14.5	330	62.5	1	1	1	1	1	1	1	1	1	1
439	26.7	27.2	15.9	341	73.8	1	1	1	1	1	1	1	1	1	1
440	24.7	7.27	17.5	347	64.7	1	1	1	1	1	1	1	1	1	1
441	24.2	3.85	15.7	348	56.6	1	1	1	1	1	1	1	1	1	1
442	24.7	3.34	14.9	363	56.9	1	1	1	1	1	1	1	1	1	1
443	21.9	3.38	15.3	345	53.1	1	1	1	1	1	1	1	1	1	1
444	23.8	3.36	14.8	334	65.9	1	1	1	1	1	1	1	1	1	1
448†	23.3	3.52	15.6	–	–										
449†	23.6	3.43	14.9	–	–										
450	23.4	3.26	14.6	338	60.3	2.01	2.01	1.01	1.01	1	1.64	1.64	1.00	1.00	1
451‡	29.1	52.5	14.7	–	–										
452	22.7	6.75	13.4	360	68.9	1	1	1	1	1	1	1	1	1	1
453	22.6	13.4	14.1	351	65.5	1	1	1	1	1	1	1	1	1	1
454	21.9	3.56	14.9	358	70.4	1	1	1	1	1	1.02	1.02	1	1	1
456	25.8	13.6	15.1	296	48.5	1	1	1	1	1	1	1	1	1	1
457	22.3	13.4	13.5	254	46.5	1	1	1	1	1	1	1	1	1	1
458	24.0	13.3	17.0	332	60.5	1	1	1	1	1	1	1	1	1	1
462	25.1	3.47	15.1	329	63.6	1	1	1	1	1	1	1	1	1	1
463	23.6	3.74	17.9	328	49.5	1	1	1	1	1	1	1	1	1	1
467†	23.5	6.97	15.9	–	–										
468†	23.0	7.05	14.1	–	–										
471†	31.9	28.8	23.1	–	–										
472†	28.8	25.9	17.0	–	–										
477†	24.3	3.67	14.6	–	–										
478	23.7	3.73	15.1	325	68.3	1	1	1	1	1	1	1	1	1	1
480	23.3	3.56	16.7	345	64.9	1	1	1	1	1	1	1	1	1	1
481	23.8	3.78	14.2	328	56.8	1	1	1	1	1	1	1	1	1	1
485	24.0	7.23	14.4	338	58.5	1	1	1	1	1	1	1	1	1	1
490	25.8	7.13	15.0	339	56.1	1.09	1.09	1	1	1	1	1	1	1	1
491	24.7	3.52	20.5	341	54.5	1	1	1	1	1	1	1	1	1	1
500†	21.8	3.46	12.6	–	–										
501	26.1	7.39	14.6	340	58.7	1.14	1.14	1.06	1	0.92	1.07	1.07	1.03	1	1.36
502	21.7	6.77	12.5	338	56.0	1.14	1.14	1.06	1	0.92	1.07	1.07	1.03	1	1.36
503	23.0	6.55	15.6	343	61.3	1.35	1.35	1.03	1	1	1.16	1.16	1.02	1	1.10
504	23.7	6.48	17.0	336	57.2	1.34	1.34	1.03	1	1	1.16	1.16	1.02	1	1.09
505	24.8	7.14	15.8	332	56.1	1.32	1.32	1.02	1	1	1.16	1.16	1.02	1	1.11
507	22.5	3.31	14.6	355	71.3	1.06	1.17	1	1	1	1.04	1.03	1	1	1
508	24.3	3.50	16.1	330	57.5	1.14	1.14	1	1	1	1.06	1.06	1	1	1
509	23.5	6.94	15.2	349	59.9	1.09	1.09	1	1	1	1.04	1.04	1	1	1
510	25.3	7.14	17.7	345	60.1	1.14	1.13	1	1	1	1.06	1.06	1	1	1
NPBench															
compute	24.6	15.5	15.3	324	74.2	1	1	1	1	1	1	1	1	1	1
crc16	22.1	3.56	47.2	347	71.8	1	1	1	1	1	1	1	1	1	1
deep_learning/conv2d_bias	144	1137	39.1	443	949	1	1	1	1	1	1	1	1	1	1
deep_learning/mlp	21.7	6.81	18.1	349	61.2	1	1	1	1	1	1.00	1.00	1	1	1
deep_learning/resnet	100	2350	51.3	443	1986	1	1	1	1	1	1	1	1	1	1
deep_learning/softmax	33.4	349	20.1	346	311	1.00	1.00	1	1	1	1.03	1.03	1	1	1
go_fast	23.6	13.7	16.5	329	67.6	1	1	1	1	1	1	1	1	1	1
polybench/atax	22.1	7.01	18.2	334	58.3	1	1	1	1	1	1	1	1	1	1
polybench/bicg	25.1	14.3	16.7	328	61.2	1	1	1	1	1	1	1	1	1	1
polybench/cholesky	27.4	57.8	18.9	337	105	1	1	1	1	1	1	1	1	1	1
polybench/covariance2	23.6	53.3	17.5	340	89.6	1.00	1	1.00	1.00	1	1.00	1	1.00	1.00	1
polybench/deriche	21.9	12.8	19.6	333	69.1	1	1	1	1	1	1	1	1	1	1
polybench/durbin	21.5	12.8	14.9	330	65.8	1	1	1	1	1	1	1	1	1	1
polybench/floyd_warshall	24.0	25.6	18.8	346	88.2	1	1	1	1	1	1	1	1	1	1
polybench/gemm	25.4	55.8	16.6	358	103	1.13	1.13	1	1	1	1	1	1	1	1
polybench/gemver	22.3	25.8	18.4	331	71.2	1	1	1	1	1	1	1	1	1	1
polybench/gesummv	25.0	14.4	19.3	340	63.8	1	1	1	1	1	1	1	1	1	1
polybench/gramschmidt	25.5	109	17.9	319	145	1	1	1	1	1	1	1	1	1	1
polybench/k2mm	26.4	115	16.3	339	127	1.19	1.19	1	1	1	1	1	1	1	1
polybench/k3mm	24.0	108	17.2	340	125	1.31	1.31	1	1	1	1	1	1	1	1
polybench/lu	23.9	52.1	15.8	340	104	1	1	1	1	1	1	1	1	1	1
polybench/ludcmp	24.8	52.0	17.2	328	116	1	1	1	1	1	1	1	1	1	1
polybench/mvt	24.7	13.9	18.2	328	59.4	1	1	1	1	1	1	1	1	1	1
polybench/symm	22.2	26.9	15.4	338	75.0	1	1	1	1	1	1	1	1	1	1
polybench/syr2k	24.4	25.9	15.6	332	65.9	1	1	1	1	1	1	1	1	1	1
polybench/syrk	24.0	13.5	14.1	346	71.3	1	1	1	1	1	1	1	1	1	1
polybench/trisolv	22.8	13.9	13.8	356	72.4	1	1	1	1	1	1	1	1	1	1
polybench/trmm	24.4	26.9	15.8	342	75.9	1	1	1	1	1	1	1	1	1	1
pythran/arc_distance	24.2	13.7	17.3	331	64.3	1	1	1	1	1	1	1	1	1	1
scattering_self_energies	147	3550	117	465	3094	1.98	1.98	1.00	1.00	1	2.52	2.52	1.00	1.00	1
weather_stencils/hdiff	34.2	109	15.2	326	127	1	1	1	1	1	1	1	1	1	1
weather_stencils/vadv	44.7	795	22.4	348	717	1	1	1	1	1	1	1	1	1	1

Table 7, continued.

	zkVM proving time (s)					Size with mechanisms disabled, over the size with all enabled									
	Core proof			Groth16		Halo2 cells					Noir gates				
Program	SP1	RISC0	Cairo	SP1	RISC0	All	Fixed	Sel.	Array	Mem.	All	Fixed	Sel.	Array	Mem.
Pythran															
allpairs (sqr_dists)	24.6	26.7	16.8	257	56.0	1	1	1	1	1	1	1	1	1	1
allpairs (sqr_dists_loops)	26.9	28.1	15.0	252	53.2	1	1	1	1	1	1	1	1	1	1
arc_distance	24.7	6.90	16.8	333	61.9	1	1	1	1	1	1	1	1	1	1
bubble_sort	21.7	3.46	16.3	342	58.2	1	1	1	1	1	1	1	1	1	1
ccopy	21.7	3.45	14.4	333	50.7	1.61	1.61	1.03	1.00	1	1.16	1.16	0.94	1.00	1.48
clip	21.2	6.38	16.7	258	45.1	1	1	1	1.00	1	1.00	1	1.00	1	1
create_grid†	24.6	13.2	–	–	–										
cronbach	22.6	6.36	16.5	327	69.5	1	1	1	1	1	1	1	1	1	1
crotg	21.7	6.37	17.3	336	55.9	1.00	1	1.00	1.00	1	1.00	1	1.00	1	1
d2q9_nxnyns	29.6	52.1	18.8	327	100	1	1	1	1	1	1	1	1	1	1
deuxd_convolution	33.2	118	17.1	372	137	1	1	1	1	1	1	1	1	1	1
euclidean_distance_square	22.8	7.04	33.7	339	62.0	1	1	1	1	1	1	1	1	1	1
fbcorr_numpy	31.5	135	27.8	350	163	1	1	1	1	1	1	1	1	1	1
fibo_seq	21.2	3.47	46.3	337	59.4	1.00	1	1.00	1.00	1	1.17	1	1.17	1.17	1
goodExpoMeasure	29.0	323	19.5	354	311	1	1	1	1	1	1	1	1	1	1
harris	24.6	3.65	15.0	330	61.1	1	1	1	1	1	1	1	1	1	1
hasting	21.5	6.86	21.2	337	57.9	1	1	1	1	1	1	1	1	1	1
insertion_sort	22.0	3.50	42.4	342	52.9	1	1	1	1	1.89	1	1	1	1	2.03
ising	21.7	3.49	13.7	285	58.8	1	1	1	1	1	1	1	1	1	1
l2norm	23.5	14.3	19.5	349	63.1	1.00	1	1.00	1.00	1	1	1	1	1	1
lap	25.4	14.2	29.1	372	78.7	1	1	1	1	1	1	1	1	1	1
lapl3d	25.2	160	15.3	335	193	1.04	1.04	1	1	1	1.13	1.13	1	1	1
ln	21.5	6.94	20.1	331	65.7	1	1	1	1	1	1	1	1	1	1
log_likelihood	25.1	28.6	14.3	344	76.9	1	1	1	1	1	1	1	1	1	1
lu (factorMatrix0)	22.1	6.82	15.1	345	61.7	1.07	1.07	1	1	1.51	1.02	1.02	1	1	4.25
lu (factorMatrix1)	22.0	7.11	20.7	336	71.1	1.03	1.03	1	1	1.45	1.01	1.01	1	1	3.88
make_decision	24.4	56.4	14.8	343	85.2	1.03	1.02	1.01	1.01	0.97	1.08	1.08	1	1	1
matmul	25.3	27.9	15.1	330	73.0	1	1	1	1	1	1	1	1	1	1
multi_export†	23.4	3.44	–	–	–										
multitype (check)	21.3	3.49	13.8	354	55.2	1	1	1	1	1	1	1	1	1	1
multitype (check2)	21.6	3.47	14.9	316	56.4	1	1	1	1	1	1	1	1	1	1
multitype (times)	22.8	3.55	23.2	330	51.3	4.81	1	4.81	4.81	1	7.32	1	7.32	7.32	1
pairwise_numpy	24.6	108	16.0	337	132	1.01	1	1.01	1.01	1	1	1	1	1	1
periodic_bc†	23.2	3.59	–	–	–										
periodic_dist	26.7	113	19.9	378	154	1	1	1	1	1	1	1	1	1	1
perm†	24.5	7.55	–	–	–										
projection_simplex	22.7	3.58	14.2	330	56.3	1.12	1.12	1	1	1	1.07	1.07	0.99	1	1
pythagorean_triples	21.3	3.48	15.5	342	56.2	1	1	1	1	1	1	1	1	1	1
reverse_cumsum	21.3	3.48	21.2	339	55.4	1	1	1	1	1	1	1	1	1	1
rosen	23.4	3.69	18.3	323	61.9	1	1	1	1	1	1	1	1	1	1
smoothing	21.2	3.49	21.8	338	48.4	1	1	1	1	1	1	1	1	1	1
specialconvolve	24.0	7.07	19.4	256	43.7	1	1	1	1	1	1	1	1	1	1
spectralDNS (add_pressure_diffusion)	26.8	14.5	16.3	335	63.7	1	1	1	1	1	1	1	1	1	1
spectralDNS (cross2)	24.5	14.3	16.6	335	61.6	1	1	1	1	1	1	1	1	1	1
sumarray3d	26.7	165	18.0	343	192	1.01	1	1.01	1.01	1	1	1	1	1	1
tax	21.4	6.33	19.8	337	68.7	1	1	1	1	1	1	1	1	1	1
value_index	21.7	6.38	15.6	348	62.7	1.85	1.85	1	1	1	1.23	1.23	1.00	1	1
vibr_energy	22.9	6.98	19.0	393	70.5	1	1	1	1	1	1	1	1	1	1
wdist	21.6	12.8	31.1	352	61.8	1	1	1	1	1	1	1	1	1	1
weights	24.0	3.84	14.2	337	58.3	1	1	1	1	1	1	1	1	1	1

Table 8. Verification time in milliseconds.

	Halo2			Noir			zkVM		
Program	Zinnia	Hand	Direct	Zinnia	Hand	Direct	SP1	RISC0	Cairo
DS-1000									
291‡	3.04	4.08	3.75	17.1	15.0	14.3	127	11.7	54.3
298	6.31	6.33	8.51	15.2	14.8	14.8	129	16.2	47.3
300	6.56	6.67	14.9	8.57	12.5	13.0	88.7	17.9	45.7
301†	7.12	5.50	7.49	16.3	18.0	14.9	138	18.5	45.9
302†	7.35	6.88	7.00	17.4	13.7	16.2	88.3	17.8	43.9
303†	9.82	6.66	5.98	15.0	13.6	14.2	134	18.5	48.0
304†	6.05	7.55	7.06	15.4	10.4	15.8	115	18.2	45.0
309	10.5	11.6	12.2	15.5	17.0	17.9	115	18.1	50.4
310	7.75	8.09	8.63	10.8	10.0	12.7	85.0	18.5	43.1
311	10.9	10.5	13.7	14.0	16.0	13.7	87.1	18.4	43.0
312	11.1	7.21	12.0	14.3	16.1	12.6	94.2	18.4	50.9
313	10.5	7.83	10.7	13.1	9.60	11.1	83.6	18.3	44.7
314	15.7	11.7	11.7	16.5	18.0	16.0	139	18.0	43.4
317†	17.4	20.5	14.4	21.7	22.4	23.3	116	20.3	46.5
318	8.34	10.2	9.62	14.8	14.7	13.4	98.1	18.5	46.2
319	13.8	14.9	27.6	17.2	19.4	18.7	99.1	13.3	44.6
320	9.12	8.71	10.5	16.1	16.0	19.9	105	18.0	44.9
321	8.77	8.51	10.2	13.6	14.1	13.0	85.5	18.5	45.2
322	6.43	6.83	7.71	14.6	16.3	13.1	127	18.1	45.0
323	5.86	5.58	6.76	13.4	14.5	12.5	82.7	18.5	44.9
324	6.02	7.02	6.31	14.0	9.63	9.59	82.0	18.7	45.8
325	7.20	9.81	9.08	16.6	14.7	10.5	85.5	11.7	47.1
326	5.97	7.05	6.80	15.5	9.67	13.5	96.7	16.3	43.1
329	12.1	9.83	11.0	9.98	14.0	12.1	119	16.5	44.4
330	11.7	12.4	13.3	16.4	18.1	17.8	100	20.0	65.8
334	11.7	11.1	12.6	13.6	18.3	15.6	85.6	19.8	44.0
335	12.0	9.70	12.6	9.21	10.1	11.0	89.2	12.4	45.3
336†	5.84	6.48	4.23	10.6	10.6	10.9	84.8	18.3	43.8
337†	5.01	4.29	4.72	14.8	10.2	9.99	92.9	18.5	43.5
338†	4.62	4.40	5.05	12.0	12.1	11.8	84.8	18.4	46.3
339†	5.32	4.84	6.59	15.9	15.7	13.8	85.1	18.1	45.6
340†	5.70	4.76	6.97	9.36	15.3	8.96	124	18.6	45.6
341†	7.38	6.73	6.56	12.3	12.0	13.6	91.6	18.0	44.0
342†	7.62	6.88	9.62	14.9	15.1	16.1	126	18.0	44.5
343†	6.60	8.09	7.06	11.1	16.7	12.0	98.2	18.0	44.9
345	6.01	6.21	6.84	10.5	12.1	9.82	129	18.1	43.1
346	10.7	10.2	10.1	17.7	13.4	13.7	90.6	18.6	57.1
347	13.0	10.1	8.40	16.3	15.0	17.6	127	19.8	44.6
349‡	2.96	3.61	3.30	13.9	14.6	13.8	95.0	18.0	45.4
353	7.22	7.36	12.1	17.7	14.7	12.3	95.6	18.1	44.6
355	9.65	11.0	10.2	12.3	19.8	13.5	96.3	18.1	47.0
356	9.57	9.42	10.6	9.40	9.70	16.4	123	18.0	43.5
357	11.9	8.67	12.8	17.1	15.6	12.8	98.7	18.3	68.8
358	10.1	11.1	10.5	17.9	13.3	13.3	133	16.4	43.7
359†	5.12	5.29	5.81	9.48	15.7	9.25	87.6	17.9	45.5
360†	5.54	7.18	5.64	16.4	10.6	16.1	83.7	18.5	44.5
361†	7.08	7.53	7.74	19.7	20.8	12.1	86.2	18.0	46.4
363	12.0	10.9	13.7	13.6	10.5	12.8	88.6	19.8	45.5
364	32.8	24.9	25.5	25.5	21.8	22.3	118	18.8	46.6
365	11.8	12.4	13.7	10.9	17.1	14.5	83.1	19.9	55.9
368	8.75	9.22	8.94	13.3	12.3	16.0	104	18.0	42.9
369	7.82	9.71	8.99	15.5	14.6	15.8	130	18.5	43.7
370	8.03	10.1	8.72	16.4	18.6	14.5	91.2	18.6	49.6
373‡	2.86	8.37	24.4	11.9	12.3	12.6	92.4	22.0	46.2
374	9.72	10.3	37.4	12.7	16.8	10.4	119	19.3	44.4
375	12.1	10.5	32.6	16.5	16.0	16.8	82.2	19.4	44.8
381	11.4	22.3	61.1	15.6	9.85	16.3	133	19.9	45.2
382	20.6	30.2	70.6	12.6	17.9	16.9	86.6	19.1	49.5
383	13.3	11.9	69.3	15.3	14.8	10.9	90.5	19.3	44.0
385†	125	136	115	78.5	91.9	90.3	95.1	42.6	63.1
386†	348	337	325	418	294	328	442	118	108
387†	125	115	117	112	95.2	127	84.1	42.4	65.6
389†	8.64	10.0	9.42	13.4	12.2	13.2	86.3	16.4	43.2
391†	10.6	7.32	10.1	17.4	15.1	13.8	86.3	19.3	44.4
392†	9.17	6.99	7.03	11.2	17.0	16.7	92.2	18.0	43.5
393†	8.61	9.35	8.51	14.8	13.6	15.0	97.0	19.1	45.9
398	7.64	5.42	7.89	17.9	12.0	12.9	96.7	18.5	44.1
399	7.15	6.93	7.09	16.0	12.6	10.4	130	18.0	45.3
400‡	2.69	3.06	3.32	11.7	11.9	12.4	88.9	11.6	43.7
401‡	3.64	3.88	4.02	14.0	15.6	14.2	94.9	18.0	43.5
402	4.26	5.61	5.09	14.8	15.4	14.9	103	18.5	43.7
403	4.10	3.43	3.84	11.0	13.7	14.1	94.1	16.5	45.6

Table 8, continued.

	Halo2			Noir			zkVM		
Program	Zinnia	Hand	Direct	Zinnia	Hand	Direct	SP1	RISC0	Cairo
409	8.97	11.7	10.5	17.7	14.8	17.5	99.6	18.2	65.8
412	6.46	6.68	6.45	15.5	11.2	13.3	128	18.5	43.5
413	7.19	6.80	9.22	14.0	9.83	12.1	91.5	15.4	50.9
414	5.80	5.28	6.65	14.5	13.5	12.2	96.9	18.0	45.7
415	6.43	6.25	7.62	11.3	12.7	12.2	127	18.5	65.6
416	11.4	7.78	9.17	13.5	17.8	14.0	88.7	19.2	45.8
417	6.39	6.84	7.37	14.9	14.4	12.6	85.9	17.9	50.0
418	7.27	8.31	6.88	12.5	16.8	12.9	84.7	19.9	45.6
419	10.5	8.18	10.4	12.1	17.1	12.8	91.2	12.4	46.9
420	6.01	6.63	6.50	14.1	19.1	15.8	92.6	18.1	46.0
422	12.3	14.2	13.3	11.6	15.6	14.8	89.8	21.5	53.4
428	7.51	6.20	10.5	12.0	10.6	11.9	100	18.2	44.9
429	8.18	7.21	8.26	13.7	15.2	11.0	91.1	17.8	43.4
430	7.31	7.49	8.73	13.0	16.7	16.2	94.3	12.6	60.7
431	7.78	7.19	8.71	15.2	13.5	14.0	96.1	13.9	44.6
433	6.66	7.85	5.88	10.2	9.81	14.7	121	17.5	44.4
434	8.20	7.75	7.12	15.4	15.9	13.6	127	15.4	43.1
435†	5.74	5.14	5.52	14.6	9.70	12.1	86.5	18.2	46.1
436	10.5	9.91	13.0	10.6	17.2	13.5	126	11.6	45.5
437	8.15	7.44	10.5	10.0	10.6	10.6	89.5	18.5	47.6
438	5.85	5.94	8.96	12.5	8.81	12.5	128	16.5	43.2
439	20.7	19.2	26.0	25.4	25.7	20.7	81.9	14.2	51.7
440	10.0	11.2	10.9	15.9	16.2	14.5	91.8	16.5	42.7
441	4.14	4.94	4.32	14.2	13.4	12.2	81.8	11.5	44.1
442	7.42	6.98	9.26	11.6	13.2	14.0	83.8	15.4	46.4
443	9.70	10.6	12.9	18.1	18.9	16.4	92.6	17.9	46.6
444	8.63	9.68	13.2	14.1	9.23	9.67	97.3	12.9	46.0
448†	6.18	7.26	6.62	13.6	14.0	16.4	84.4	17.9	45.7
449†	6.63	9.03	8.53	18.9	17.1	17.1	125	15.3	43.9
450	13.9	13.9	19.1	12.4	15.3	14.7	110	12.9	51.4
451‡	3.53	4.04	3.58	13.8	13.3	13.8	88.6	22.8	50.2
452	10.4	9.39	10.6	13.8	15.5	12.7	92.5	19.4	45.2
453	8.54	7.18	8.01	10.4	12.3	10.1	95.5	20.6	48.5
454	10.5	11.1	11.5	13.2	12.8	14.1	98.7	13.0	47.7
456	12.2	11.6	10.5	19.2	17.3	17.5	84.3	20.5	45.1
457	10.5	10.6	13.8	15.4	15.8	13.6	87.1	20.2	46.2
458	9.66	12.4	12.1	16.8	17.6	18.0	90.4	13.2	45.6
462	8.77	8.03	12.8	18.9	10.3	11.0	140	12.9	45.5
463	8.41	5.58	12.3	14.4	14.2	14.7	97.1	18.1	44.2
467†	5.65	6.43	5.64	10.9	15.8	12.9	88.7	19.3	44.5
468†	5.62	7.19	4.61	15.3	13.7	12.9	83.0	19.2	46.8
471†	25.6	26.0	33.9	17.7	22.0	24.6	85.8	21.5	48.5
472†	27.6	23.3	24.3	15.4	16.7	20.1	87.1	21.5	50.1
477†	5.02	6.29	5.95	17.8	17.8	13.6	81.9	17.5	58.4
478	4.65	4.92	5.94	12.2	12.4	12.3	85.9	18.1	45.2
480	6.14	6.24	9.25	17.4	15.7	17.7	88.1	18.7	44.7
481	7.12	7.87	9.18	17.0	14.7	16.7	95.7	18.5	45.6
485	11.1	9.31	9.27	10.6	9.41	9.76	81.4	12.5	45.8
490	16.2	19.9	160	16.5	15.8	15.6	117	19.4	46.1
491	7.05	9.81	8.25	17.4	15.7	15.7	94.5	18.0	46.6
500†	10.0	6.86	10.9	12.9	12.6	11.5	89.2	18.0	46.1
501	11.9	15.4	18.2	16.0	14.6	17.5	96.5	19.3	44.4
502	11.8	14.8	16.5	11.3	11.0	15.2	84.2	12.5	45.4
503	14.0	16.8	25.0	15.8	12.5	13.5	99.4	13.9	45.3
504	16.8	22.7	21.9	12.1	10.1	14.2	114	13.9	45.4
505	17.1	25.6	23.3	18.3	15.1	15.1	122	19.5	69.9
507	8.91	7.84	15.4	13.9	16.0	15.3	94.8	15.4	48.1
508	11.3	9.00	49.9	14.0	15.0	14.5	91.0	18.4	46.5
509	11.2	11.2	35.9	15.0	17.6	17.3	90.8	13.8	45.8
510	12.8	8.76	32.5	15.0	10.3	14.7	101	13.9	43.5
NPBench									
`compute`	15.5	15.3	16.6	10.6	14.0	15.3	86.9	20.4	49.1
`crc16`	41.3	37.8	37.5	13.2	15.5	14.4	93.6	18.4	70.3
`deep_learning/conv2d_bias`	57.9	60.7	81.6	46.8	47.0	44.5	133	164	54.8
`deep_learning/mlp`	9.76	10.7	13.2	14.8	13.9	13.1	85.2	19.3	44.0
`deep_learning/resnet`	63.8	113	128	21.7	27.5	30.8	102	371	51.8
`deep_learning/softmax`	72.9	119	125	19.4	22.1	20.6	131	59.7	45.9
`go_fast`	14.5	19.4	19.6	17.0	15.2	16.8	83.4	16.4	44.5
`polybench/atax`	12.1	11.1	12.2	14.6	15.3	15.5	127	19.3	57.8
`polybench/bicg`	9.41	10.0	10.7	14.8	13.8	15.1	91.1	21.0	50.6
`polybench/cholesky`	15.7	14.4	18.5	10.8	15.2	14.4	98.1	16.4	87.9
`polybench/covariance2`	13.4	14.1	21.5	15.2	15.2	17.0	95.8	19.6	45.6
`polybench/deriche`	25.7	36.3	37.3	14.9	14.5	16.5	106	14.7	45.2

Table 8, continued.

Program	Halo2			Noir			zkVM		
	Zinnia	Hand	Direct	Zinnia	Hand	Direct	SP1	RISC0	Cairo
polybench/durbin	8.60	8.79	12.5	11.3	14.2	11.8	95.3	14.7	46.1
polybench/floyd_warshall	50.9	66.0	47.4	14.0	14.5	16.0	94.6	20.1	47.8
polybench/gemm	24.0	27.2	29.9	14.2	17.5	17.3	132	16.5	45.4
polybench/gemver	16.7	21.0	27.1	18.3	19.2	14.7	94.0	21.5	49.8
polybench/gesummv	18.1	23.1	20.9	15.2	15.1	14.6	140	13.3	48.6
polybench/gramschmidt	25.5	29.7	34.9	15.6	14.0	14.4	90.5	27.9	47.8
polybench/k2mm	27.1	30.8	30.5	16.4	15.7	16.3	99.6	17.5	58.0
polybench/k3mm	21.2	28.3	36.9	14.4	14.8	11.4	95.1	17.4	47.5
polybench/lu	15.2	15.6	15.8	12.8	13.6	12.4	94.5	16.4	44.8
polybench/ludcmp	16.1	17.6	18.0	12.3	9.91	12.4	99.3	18.4	45.1
polybench/mvt	8.54	11.1	13.1	14.2	12.1	15.2	124	13.2	45.3
polybench/symm	20.3	20.1	17.2	11.9	12.2	12.0	94.8	21.5	54.4
polybench/syr2k	26.2	23.6	34.3	13.4	13.6	13.8	90.3	21.7	35.3
polybench/syrk	33.7	30.5	30.0	18.5	17.8	13.4	118	20.4	48.2
polybench/trisolv	12.3	12.3	9.46	12.2	15.3	14.7	86.9	20.7	48.1
polybench/trmm	16.4	16.7	20.0	17.6	16.7	10.7	100	22.1	44.8
pythran/arc_distance	19.2	24.7	24.7	14.7	10.3	14.2	88.9	20.5	46.3
scattering_self_energies	607	245	791	19.1	17.3	18.6	138	520	54.5
weather_stencils/hdiff	48.4	55.6	62.4	20.4	17.7	17.1	86.9	15.8	45.3
weather_stencils/vadv	93.7	110	121	29.3	17.8	24.7	90.6	131	49.1
Pythran									
allpairs (sqr_dists)	47.0	52.0	43.2	14.1	15.1	17.4	126	19.7	44.6
allpairs (sqr_dists_loops)	38.6	46.0	43.4	14.4	15.6	14.4	126	21.4	49.6
arc_distance	13.4	14.1	12.7	13.8	15.6	13.8	96.6	19.3	55.5
bubble_sort	6.78	8.17	11.3	10.2	12.6	10.8	97.0	15.3	59.6
ccopy	56.4	60.6	100	14.1	13.4	12.4	90.7	12.9	65.5
clip	10.4	8.65	10.3	16.7	12.2	16.9	101	16.5	80.6
create_grid†	11.8	13.3	11.2	15.6	14.2	18.5	100	20.4	–
cronbach	8.86	9.44	9.70	17.1	12.6	11.4	95.1	13.8	55.7
crotg	9.85	10.0	8.07	14.2	11.1	16.6	101	17.9	46.9
d2q9_nxnyns	40.0	43.5	53.1	26.1	25.0	26.3	101	22.3	45.4
deuxd_convolution	18.0	19.9	57.0	14.5	19.0	13.1	115	24.6	43.9
euclidean_distance_square	7.69	7.07	6.96	13.6	12.8	14.6	89.8	19.2	55.3
fbcorr_numpy	87.7	71.2	72.0	18.1	18.7	16.9	81.7	40.5	45.5
fibo_seq	19.3	24.4	24.5	14.0	14.8	15.0	90.4	17.3	57.6
goodExpoMeasure	60.9	89.6	52.6	14.4	17.2	16.2	87.7	59.5	44.4
harris	8.44	8.46	10.7	13.6	14.6	14.0	118	11.6	43.7
hasting	5.11	6.45	7.78	14.0	17.5	9.05	81.9	19.8	91.6
insertion_sort	9.90	7.73	16.1	14.2	14.9	14.5	90.3	17.9	50.5
ising	8.51	8.74	7.10	11.9	12.1	11.9	92.6	17.8	47.2
l2norm	15.1	15.1	16.7	15.5	15.7	13.6	93.0	20.3	50.5
lap	11.0	12.0	9.96	15.4	15.1	15.9	92.3	21.0	45.5
lapl3d	18.8	21.8	33.4	14.6	15.4	15.3	94.6	29.9	48.9
ln	9.72	13.1	12.3	10.00	15.2	11.0	85.8	19.3	59.6
log_likelihood	15.3	23.0	16.1	17.7	13.2	13.6	128	21.5	48.7
lu (factorMatrix0)	13.6	12.3	14.1	19.0	12.9	13.5	89.2	19.3	47.5
lu (factorMatrix1)	9.84	8.57	12.2	17.2	15.1	14.0	111	18.2	47.9
make_decision	17.1	20.3	16.1	14.3	14.7	15.8	93.4	22.7	43.3
matmul	37.8	41.2	40.3	15.3	16.0	15.4	83.3	21.5	49.4
multi_export†	5.21	4.02	4.38	13.2	12.8	12.7	88.6	18.0	–
multitype (check)	6.60	5.57	5.46	14.9	12.8	18.4	94.9	18.0	44.6
multitype (check2)	6.24	6.36	5.33	18.0	15.1	13.6	101	18.1	56.4
multitype (times)	4.39	4.00	4.87	13.0	17.4	14.3	81.5	17.2	63.5
pairwise_numpy	27.7	29.6	26.7	11.0	15.8	14.3	91.6	24.0	47.2
periodic_bc†	4.56	5.78	6.34	12.6	15.3	15.8	93.8	18.2	–
periodic_dist	30.7	32.8	454	21.5	17.6	20.7	87.1	20.5	44.3
perm†	5.09	5.87	4.17	13.5	17.1	14.2	122	19.2	–
projection_simplex	9.73	11.1	11.6	12.8	14.5	16.1	128	17.9	48.2
pythagorean_triples	3.32	4.66	4.46	11.5	8.84	14.9	95.1	16.4	46.0
reverse_cumsum	5.41	6.10	5.92	13.5	15.2	14.2	97.1	18.5	63.1
rosen	8.56	8.13	8.68	11.6	11.8	12.1	140	18.1	49.0
smoothing	8.63	7.06	7.50	16.0	9.56	14.7	101	18.5	78.6
specialconvolve	12.0	9.77	10.5	17.5	14.9	15.2	81.7	19.7	62.7
spectralDNS (add_pressure_diffusion)	23.4	18.3	37.1	12.3	14.2	11.9	98.2	20.4	48.3
spectralDNS (cross2)	22.2	23.9	25.7	16.6	16.7	18.5	87.1	20.3	45.3
sumarray3d	28.5	32.4	33.3	14.7	10.9	12.0	115	42.0	56.5
tax	12.0	13.3	13.8	14.5	13.0	17.5	95.6	13.8	49.7
value_index	10.2	14.0	16.8	12.0	10.3	18.2	94.4	19.2	46.1
vibr_energy	12.3	14.0	15.1	12.4	13.6	13.3	98.3	16.5	47.1
wdist	12.0	10.0	11.1	12.5	15.4	15.9	91.2	20.3	51.1
weights	6.28	7.15	8.23	8.40	12.8	13.6	105	18.0	44.3

Received 17 March 2026