Evaluating Compiler Optimization Impacts on zkVM Performance

Thomas Gassmann
ETH Zurich
Switzerland

Stefanos Chaliasos
UCL Centre for Blockchain Technologies
United Kingdom
zkSecurity
USA

Thodoris Sotiropoulos
ETH Zurich
Switzerland

Zhendong Su
ETH Zurich
Switzerland

Abstract

Zero-knowledge proofs (ZKPs) are the cornerstone of programmable cryptography. They enable (1) privacy-preserving and verifiable computation across blockchains, and (2) an expanding range of off-chain applications such as credential schemes. Zero-knowledge virtual machines (zkVMs) lower the barrier by turning ZKPs into a drop-in backend for standard compilation pipelines. This lets developers write proof-generating programs in conventional languages (e.g., Rust or C++) instead of hand-crafting arithmetic circuits. However, these VMs inherit compiler infrastructures tuned for traditional architectures rather than for proof systems. In particular, standard compiler optimizations assume features that are absent in zkVMs, including cache locality, branch prediction, or instruction-level parallelism. Therefore, their impact on proof generation is questionable.

We present the first systematic study of the impact of compiler optimizations on zkVMs. We evaluate 64 LLVM passes, six standard optimization levels, and an unoptimized baseline across 58 benchmarks on two RISC-V–based zkVMs (RISC Zero and SP1). While standard LLVM optimization levels do improve zkVM performance (over 40%), their impact is far smaller than on traditional CPUs, since their decisions rely on hardware features rather than proof constraints. Guided by a performance impact analysis, we *slightly* refine a small set of LLVM passes to be zkVM-aware, improving zkVM execution time by up to 45% (average +4.6% on RISC Zero, +1% on SP1) and achieving consistent proving-time gains. Our work highlights the potential of compiler-level optimizations for zkVM performance and opens new directions for zkVM-specific passes, backends, and superoptimizers.

1 Introduction

Zero-knowledge proofs (ZKPs) [27] let a prover convince a verifier that a statement holds, optionally, without revealing anything beyond its truth. After decades as a theoretical concept, advances in proof systems have made ZKPs practical at scale [25, 28, 44]. Still, two obstacles have thwarted adoption: (1) developers had to handcraft low-level arithmetic circuits [9], which is both hard and error-prone [16], and (2) proving remains costly despite recent progress [19].

Zero-knowledge virtual machines (zkVMs) [6, 13, 26, 34] address both issues by enabling developers to reuse well-established languages and toolchains. This is achieved by compiling ordinary programs (e.g., Rust) to a standard ISA such as RISC-V [56], then emulating the binary to record an execution trace that a zkVM back-end proves. This design leverages tools such as LLVM [38], avoids manually-written circuits, and benefits from techniques such as GPU proving [40, 50], recursion [32, 57], and precompiles for heavy primitives. zkVMs are already deployed in key blockchain applications [35, 43], and are increasingly adopted in off-chain systems, such as anonymous credentials and client-side proving [4]. A prime example is Ethereum's long-term roadmap which is heavily centered around zkVMs. Current proposals include replacing the EVM with RISC-V to accelerate proving [15], advancing formal-verification efforts to harden zkVM correctness [55], and initiatives such as ETHProofs that monitor progress across both fronts [21].

Despite this progress, performance remains the central hurdle and a competitive factor among zkVM vendors [21, 29]. zkVMs' architecture adopts an execution model fundamentally different from traditional CPUs, where every instruction becomes a set of constraints to be proved. Yet, compilers still optimize for hardware features that are not relevant to zkVMs, such as caches, out-of-order execution, and instruction-level parallelism. All these raise the following question: *How do compiler-level techniques (i.e., optimizations) influence zkVM performance, and to what extent?*

This open question motivates this study, which aims to investigate how existing optimizations influence zkVM performance and identify concrete opportunities for improvement. We seek answers for the following three research questions:

RQ1 How do standard compiler optimizations impact zkVM performance when applied individually? (Section 4.1)
RQ2 Are there combinations of optimizations that yield significant performance improvements or degradations? (Section 4.2)

RQ3 How does the impact of optimizations on zkVMs compare to traditional architectures? (Section 4.3)

To answer these research questions, we benchmark 58 programs on two zkVMs, RISC Zero [13] and SP1 [34], across 71 optimization profiles that include 64 individual LLVM passes, six default optimization levels (e.g., -O3), and an unoptimized baseline. The selected zkVMs are the most mature and widely used ones, targeting a conventional ISA that leverages well-established toolchains [37, 58], such as LLVM. Our benchmarks cover a wide range of programs, including cryptographic primitives and mathematical computations. The effects of optimizations are quantified through three metrics: cycle count, zkVM execution time, and proving time.

Contributions. We make the following contributions:

- We present the first systematic study of how traditional compiler optimizations (both individual passes and their combinations) affect zkVM performance, and compare the results to traditional CPUs (Section 4).
- We identify and empirically validate key cost components that drive zkVM performance, that is, dynamic instruction count, and paging cycles. (Section 5.1).
- We analyze selected optimizations based on our cost components, and derive four optimization principles, which we validate empirically. We then apply these principles to refine existing passes for better zkVM performance. (Sections 5.2 and 6.1).
- We enumerate the implications of our findings, and discuss potential future directions on improving zkVM performance (Section 6.2).

Summary of findings. Some of our representative findings are: (1) default optimization levels (e.g., -O3) improve both zkVM execution and proving time by over 40%; (2) however, these gains are significantly smaller than on traditional CPUs, as many individual passes exhibit opposing effects, mainly due to wrong decision heuristics for inlining, or loop unrolling. These heuristics rely on exploiting hardware-specific features (e.g., cache locality) rather than minimizing the number of the executed instructions; and (3) autotuning proves to be an effective approach for performance-critical zkVM programs (up to 2.2× speedup compared to -O3).

Implications. We identify root causes of inefficiencies in certain passes and implement *slight* modifications to an existing compiler toolchain (i.e., LLVM). The -O3 level in our modified LLVM outperforms the original LLVM's -O3 in most benchmarks, achieving up to 45% faster zkVM execution (avg. +4% on RISC Zero; +1% on SP1) while also delivering consistent improvements in proving time. Since zkVM execution and proving are orders of magnitude more expensive than execution on traditional CPUs, even small percentage improvements can translate into minutes of savings (Section 2).

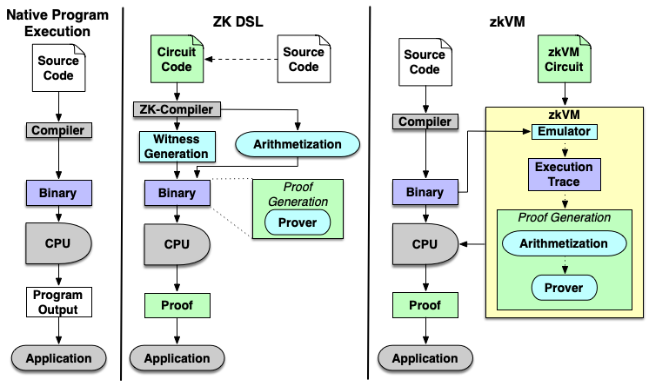


Figure 1. Architecture comparison between native execution, ZK DSLs, and zkVMs.

2 Background

Zero-Knowledge Proofs and zkVMs. A ZKP [27] is a cryptographic primitive that allows a prover to convince a verifier that a statement is valid, optionally, without revealing anything beyond its truth. A ZKP provides *completeness*, *soundness*, and optionally *zero knowledge*. In the past decade, adoption of ZKPs have accelerated with SNARKs [28, 44], which provide non-interactive, succinct proofs with fast verification and practical proving costs. ZKPs have enabled many novel applications [20, 39] such as private transactions [11] and credential systems [7, 24].

SNARKs target *circuit satisfiability* [44], representing statements as *arithmetic circuits*. Early ZK development used low-level ZK DSLs [9], where a compiler *arithmetizes* the circuit into a concrete constraint system (e.g., PLONK-style [25]) and produces a *witness generator*. Then, the *prover* uses the constraints and witness to *produce a proof*. Writing low-level ZK circuits is not only error-prone [16], but it also requires significant expertise. *Zero-knowledge virtual machines* (zkVMs) [12] raise the abstraction by compiling source code to a target ISA (often RISC-V [56]) [13, 34]. Then, the zkVM's *emulator* executes the binary and records an *execution trace* in the proper arithmetization, and the back-end produces a proof. Developers use well-established compilation toolchains (e.g., LLVM), and the zkVM exposes *precompiles* for heavy primitives (e.g., SHA-2) to replace thousands of instructions with optimized circuits. To make zkVMs practical for computation-heavy applications (e.g., ZK-Rollups [8]), much of the literature focuses on improving both execution and proof generation [6, 13, 26, 34, 49]. Figure 1 contrasts native execution, ZK DSLs, and zkVMs. For an in-depth introduction of ZKPs, we refer the reader to [51].

zkVMs vs. traditional architectures. When generating the execution trace and proving it on a zkVM, the execution model differs significantly from traditional CPU architectures, such as x86 or ARM. Most instructions have near-uniform cost, and there is no instruction latency due to cache

```
// unoptimized version
example::div:
        li t0, 8
        div a0, a0, t0
        ret
```

```
// optimized version
example::div:
        srai a1, a0, 31
        srli a1, a1, 29
        add a0, a0, a1
        srai a0, a0, 3
        ret
```

(a) CPU-oriented instruction selection can hurt zkVMs: on RISC Zero, the 'optimized' form takes 8 cycles vs. 4 unoptimized

```
/* unoptimized version */
int i, a[N], b[N];
for (i = 0; i < N; i++) {
    a[i] = 1;
    b[i] = 2;
}
```

```
/* optimized version */
int i, a[N], b[N];
for (i = 0; i < N; i++) {
    a[i] = 1;
}
for (i = 0; i < N; i++) {
    b[i] = 2;
}
```

(b) *Loop fission* often helps CPUs but can hurt zkVMs by duplicating loop-control work. N is set to 1048576.

Figure 2. Optimizations with negative effects on zkVMs.

hits/misses and other micro-architectural effects. Memory access is relatively cheaper on zkVMs (cache misses are not penalized), but page-ins and page-outs can be significantly more expensive. Further, there is no parallelism or multi-threading, and floating-point operations are not supported natively, so they need to be emulated, which makes them extremely costly. We detail a full list of differences in Appendix A. Notably, zkVM execution and proving are orders of magnitude slower than native execution (milliseconds vs. seconds to hours, cf. Figure 15 Appendix A).

Compiler optimizations and zkVMs. Compiler optimizations are semantics-preserving transformations that aim to improve speed, reduce size, or enhance other attributes of code [3, 54]. We present two motivating examples where optimizations that aid conventional CPUs degrade zkVM performance, exposing a mismatch between hardware-oriented assumptions and the zkVMs' proof-centric cost model.

Strength reduction. LLVM's *strength reduction* replaces division instruction with a shift-and-add sequence (Figure 2a). On conventional CPUs, this is beneficial because division instruction is often costlier than the equivalent bitwise instructions [23]. Our x86 measurements show that the transformed code is 3.5× faster. On zkVMs, the effect reverses, notably proving is 40% slower in RISC Zero [13]. The reason is the cost model mismatch: on zkVMs, every operation contributes to the number of constraints in the proof, and all instructions have roughly uniform cost. Replacing a single division with multiple bitwise operations therefore increases the constraint count, making proving slower.

Loop fission is an optimization that splits a loop into multiple ones to enhance locality and reduce cache misses (Figure 2b). On x86, the transformed code is ~8% faster. On zkVMs, the effect flips: duplicating loop control increases proof constraints, raising proving time by 5% on SP1 [34]. This underscores that cache-centric assumptions do not transfer to zkVMs' model.

3 Methodology

Selection of zkVMs. We evaluate two zkVMs targeting RISC-V [56] (SP1 [34] and RISC Zero [13]), chosen for maturity and widespread use in production. We also considered zkVMs with non-standard ZK-oriented ISAs such as Cairo [26] and Valida [53], but these stacks bypass existing compiler infrastructures and rely on less stable transpilers.

The selected zkVMs can directly run C and Rust programs that are lowered to RISC-V via an existing compiler toolchain, in particular LLVM [38]. Therefore, our work specifically investigates the impact of traditional compiler optimizations *implemented in LLVM* on zkVMs.

3.1 Metrics

We evaluate the impact of optimizations on zkVMs using three zkVM metrics: *cycle count*, *zkVM execution time*, and *proving time*. *Cycle count* is the total cycles of the program run on the zkVM (also known as the *guest* program), and is analogous to CPU cycles. It captures only the cost of program execution and proving, and does not include other one-time setup or initialization overheads, such as prover initialization. *zkVM execution time* is the wall-clock time for the executor to replay the guest and produce the full execution trace. *Proving time* is the wall-clock time for the prover to generate a proof and is typically larger than execution time. For native runs (RQ3), we measure native execution time to contrast optimization effects on zkVMs and traditional architectures.

3.2 Benchmark Selection

We combine standard benchmarks along with zkVM-focused workloads to cover both large, optimization-sensitive programs and common zkVM payloads. In total, our benchmark suite consists of 58 programs written in Rust and C/C++. We use PolyBench [22, 45], NPB [18, 41], and SPEC CPU 2017 [14], which include compute and memory-intensive programs. From industry benchmark suites, we include the a16z crypto zkVM, the Succinct Labs, and the RSP benchmarks [2, 33, 36]. These capture typical cryptography-heavy workloads. We further add targeted programs, such as `regex-match`, `sha256`, and `loop-sum`, to exercise loop-heavy, memory-heavy, and math/crypto patterns frequently seen in zkVMs. For the complete list of the benchmarks see Appendix B.

3.3 Analyzing Optimizations

To measure the impact of individual optimizations as well as standard optimization levels, we evaluate a total of 71 optimization profiles, which consist of 64 individual LLVM optimization passes, 1 unoptimized *baseline* profile, and 6 preset optimization levels (`-O0`, `-O1`, `-Oz`, etc.). For all 64 individual optimization passes and the `baseline` profile, we also disable Rust's MIR optimizations by setting `mir-opt-level` to `0`. The `baseline` profile has no optimizations applied and serves as a reference point for other optimization profiles.

To answer the RQs outlined in Section 1, we first measure each pass in isolation on RISC Zero and SP1 across our benchmark suite, collecting cycle count, executor time, and proving time (Section 4.1) and quantify the performance losses/gains compared to the baseline. Next, we explore optimization combinations via genetic autotuning with OpenTuner [5] (Section 4.2). Then, for RQ3, we run the same optimization profiles on traditional CPU (i.e., x86) and compare the speedups with those obtained from zkVMs to pinpoint opportunities for improvement and understand which optimizations are worth revisiting or redesigning in the zkVM context (Section 4.3).

Setup. x86 and zkVM *execution* experiments run on an AMD EPYC 7742 with 500GB of RAM on Ubuntu 22.04.5. *Proving* runs on an AMD EPYC 7542 with 32GB RAM on Ubuntu 24.04.2, equipped with an NVIDIA GeForce RTX 4090 (24GB VRAM). The SP1 prover is executed locally via its RPC API. We use SP1 4.1.4 (moongate-server 4.1.0), RISC Zero 1.2.4, Clang 19.1.7, LLVM 19.1.7, and Rust 1.85.0. For both zkVMs we used their default STARK-based proof systems.

3.4 Threats to Validity

Internal validity. Proving time on SP1 exhibits higher variance than on RISC Zero, partly due to the closed-source prover accessed via RPC and the associated network overhead. To mitigate this, We use at least 10 samples per configuration and increase the sample size for noisy cases.

External validity. Our proving time measurements are accelerated on an NVIDIA RTX 4090. However, different GPUs or CPUs may change effects. We study two popular RISC-V zkVMs, but other zkVMs or designs (e.g., different ISAs) may behave differently. Extending the study to additional zkVMs is left for future work. Our RQ3 comparison is limited to x86. While ARM or RISC-V CPUs could provide additional insights, we choose x86 since it is currently the most widely used architecture for running zkVM workloads. Finally, results may not generalize to all workloads, but we mitigate this by selecting benchmarks spanning scientific, loop-intensive, and cryptographic domains.

Construct validity. Several benchmarks use reduced input sizes to keep proving feasible. We evaluate 64 individual LLVM passes essentially in isolation with default parameters. Many additional passes and tune strategies exist, and some benefits only appear in specific combinations. Although we explored combinations via autotuning, the whole phase-ordering space remains intractable.

Table 1. Number of instances where optimized code leads to performance gains or losses in execution and proving time.

	Execution		Proving	
zkVM	**Gain (>2%)**	**Loss (<-2%)**	**Gain (>2%)**	**Loss (<-2%)**
RISC Zero	370	437	302	241
SP1	314	124	347	174

4 Results

In this section, we present the results for each research question outlined in Section 1.

4.1 RQ1: Impact of Individual LLVM Passes

Figure 3 presents the top 25 LLVM passes with the highest average impact (either positive or negative) across all benchmarks, zkVMs, and metrics. Each result reflects the performance difference relative to an unoptimized baseline with no passes applied. The remaining 39 passes have minimal impact and are omitted for brevity, as they change execution time and proving time by only 0.9%–1%, on average.

zkVM execution time. Only a few passes, on average, improve zkVM execution time when applied individually. Out of 64 passes, 12 reduce execution time on RISC Zero by at least 1% on average, and 13 do so on SP1. The most harmful pass on both zkVMs is `licm`, which increases execution time by 11.8% on RISC Zero, and 7.1% on SP1. Section 5 analyzes the root causes of this regression in detail.

Inlining is beneficial for both zkVMs: the `inline` and `always-inline` passes lead to performance gains of 28.4% and 5.7% on RISC Zero, and 19.3% and 5.3% on SP1, respectively. Other passes, such as `instcombine` or `sroa`, perform well on some benchmarks, while degrading performance on others. Both zkVMs exhibit similar trends, with no pass consistently having opposite effects on RISC Zero and SP1.

Proving time. Proving time closely follows the trends observed in zkVM execution. The most beneficial pass is again `inline` (22.4% performance gain on both SP1 and RISC Zero). The most harmful pass remains `licm`, which increases proving time by 8.4% on SP1 and 13.5% on RISC Zero.

Cycle count. Similar trends are observed when examining cycle count. Inlining reduces cycle count with a 30.8% reduction on SP1 and a 29.9% on RISC Zero. The pass that increases the cycle count the most is `licm`, which explains its negative effect on zkVM execution and proving.

Categories of effects. For completeness, Figure 4 shows the number of benchmarks where each pass causes: (1) moderate performance loss (-5% to -2%), (2) severe loss ($\leq -5\%$), (3) moderate gain (2% to 5%), or (4) severe gain ($\geq 5\%$). Inlining consistently improves both execution and proving times, though a few benchmarks still see execution slowdowns (more details in Section 5). In contrast, loop optimizations (e.g., `loop-extract`, `licm`, `loop-deletion`, `loop-unroll`) often harm zkVM performance, especially on RISC Zero.

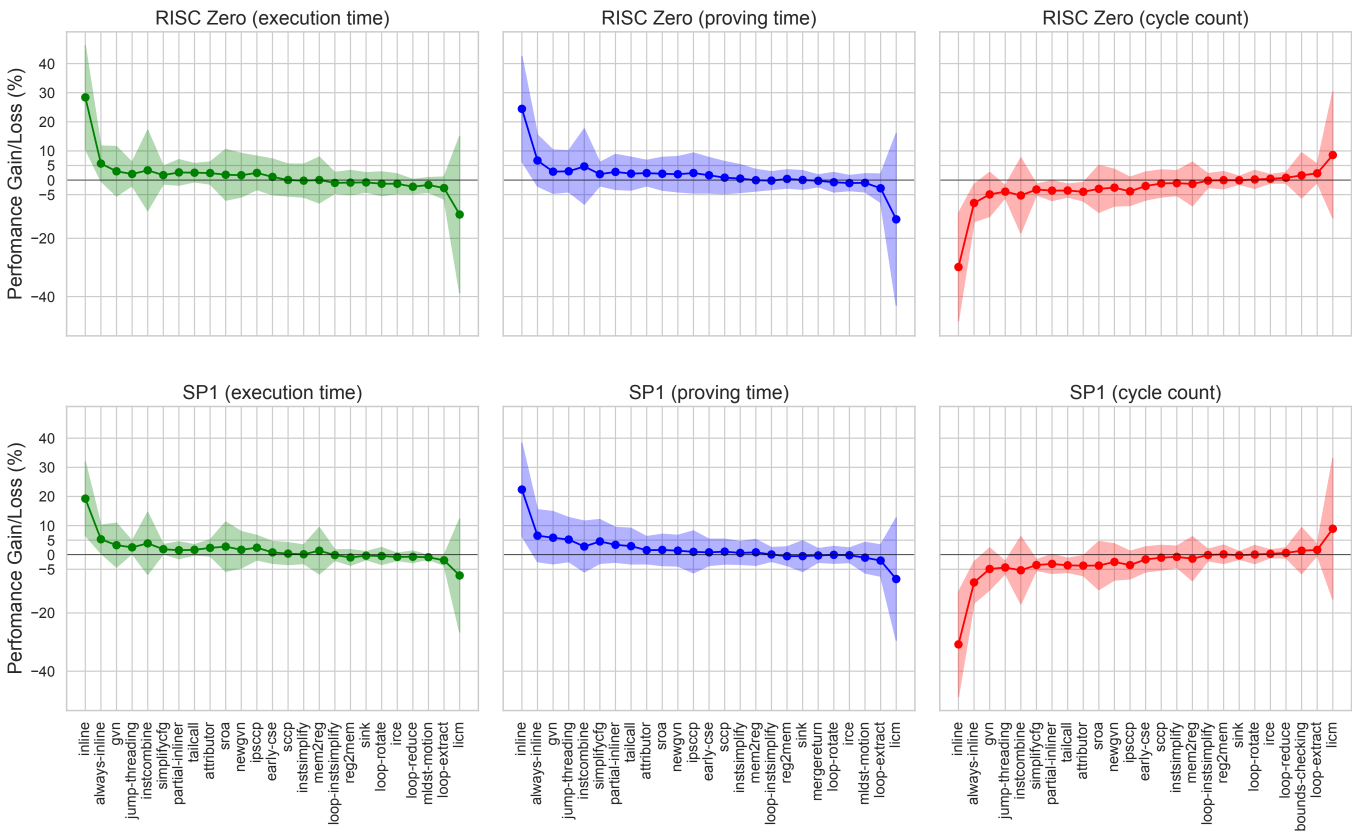


Figure 3. The impact of top 25 individual LLVM passes across all benchmarks, zkVMs (RISC Zero, SP1), and metrics (zkVM execution time, proving time, cycle count). Lines represent average impact; shaded areas show standard deviation. Higher values indicate better performance for proving time and zkVM execution time, while lower is better for cycle count.

This is because loop optimizations in LLVM operate in Loop Closed SSA (LCSSA) form, where values defined in a loop but used outside it are routed through `phi` nodes at the loop exit. This transformation often requires recomputing addresses (e.g., via `getelementptr`) or adding load/store instructions, especially when iterating over arrays, leading to extra memory operations that harm zkVM performance (Section 5). Finally, some passes, such as `instcombine` on RISC Zero execution, show a balanced impact, with similar numbers of benchmarks improving and regressing. This suggests that certain rewrites in `instcombine` may hurt zkVM performance and should be adjusted to better align with zkVMs' model (Figure 2a).

Differences in zkVMs. Differences between zkVMs are evident in Table 1, which shows how often optimized code results in performance gains or losses in execution and proving time for each zkVM. Overall, individual optimizations appear to have a more detrimental effect on RISC Zero, especially in execution, where degradation occurs roughly 4× more often compared to SP1. This could stem from differences in IRs and constraint-level optimizations across zkVM.

4.2 RQ2: Combinations of Optimizations

Standard optimization levels. Figure 5 shows the impact of standard optimization levels (`-0x` flags) on execution and proving time across both zkVMs, relative to our baseline where no optimizations are applied, including Rust's MIR optimizations (Section 3.3). For brevity, the effects on cycle count are omitted. Excluding `-O0`, no `-0x` level negatively impacts performance relative to the baseline. In contrast, `-O0` (i.e., Rust's MIR optimizations) causes regressions in 19 programs on RISC Zero and 9 on SP1. On average, standard optimization levels improve execution time by 60.5% on RISC Zero and 47.3% on SP1, and proving time by 55.5% and 51.1%, respectively. The `-O3` level consistently achieves the highest average performance gains, while `-Oz` (reducing binary size) shows the lowest.

Autotuning. Overall, the existing `-0x` flags have a positive effect on both zkVM execution and proving time. However, can we identify sequence of passes that outperform them? To explore this, we employ OpenTuner [5], a genetic-based tuning framework that finds an optimal combination of compilation flags guided by a specific fitness function. Although zkVM execution and proving time are the ultimate metrics

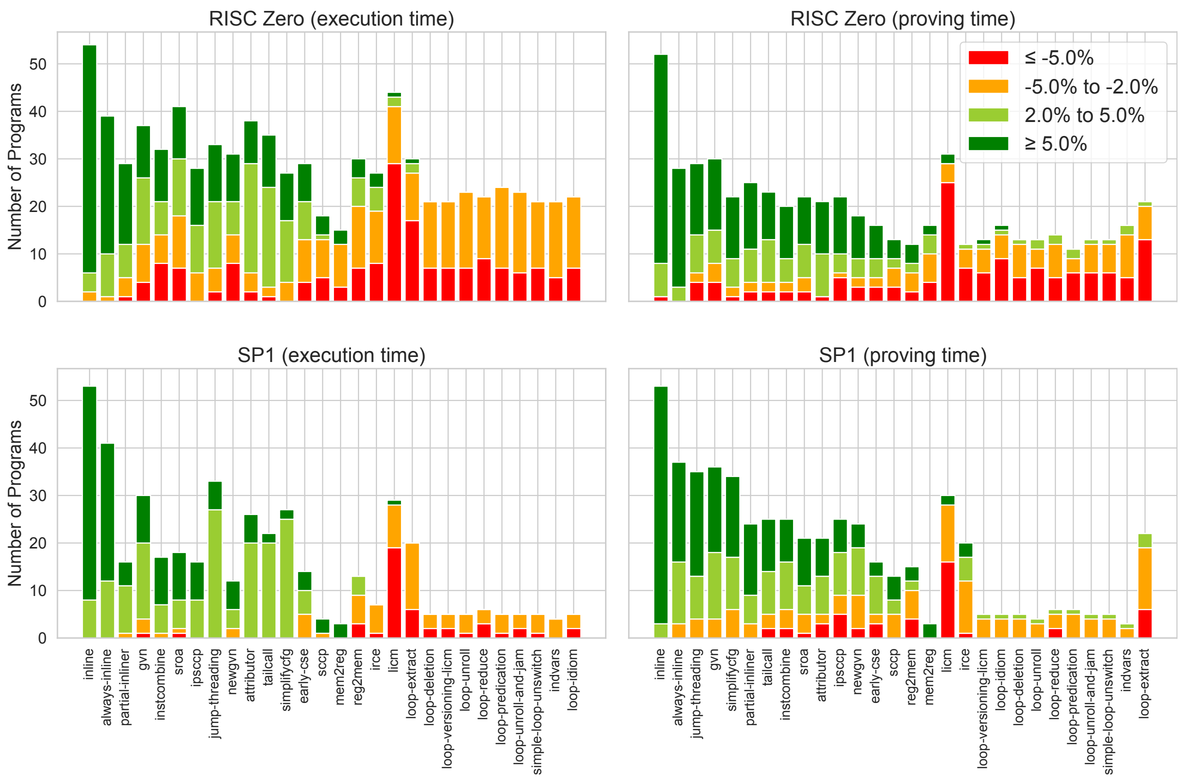


Figure 4. Counting the cases where each pass leads to (1) severe performance gain (≥ 5%), moderate performance gain (between 2% and 5%), moderate performance loss (between -2% and -5%), and severe performance loss ($\leq -5\%$).

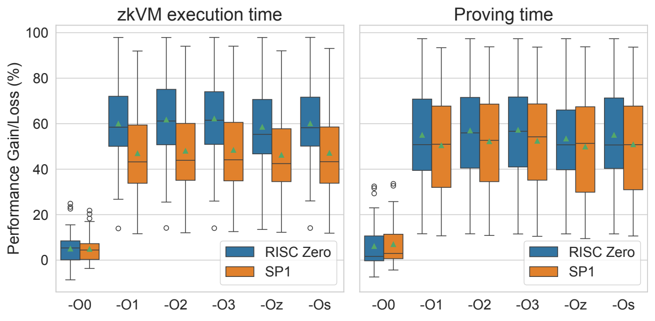


Figure 5. Impact of standard optimization levels for zkVMs.

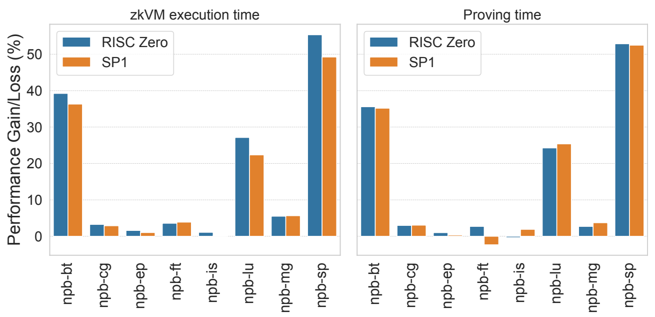
of interest, they are too expensive to evaluate repeatedly during autotuning. Instead, we use cycle count as a proxy, as it is a natural performance predictor. This choice offers two key advantages: (1) cycle count avoids measurement noise, such as the inherent imprecision of wall-clock timing and (2) it is fast to compute (~1 second on average).

We autotune each benchmark independently for each zkVM. We consider all LLVM passes as tunable parameters, creating sequences of passes up to a depth of 20. For compiler options that receive integer or enum values (e.g., `-inline-threshold`), we define appropriate tuning range and use OpenTuner's parameter types to select the optimal value. For most flags, these ranges are large enough to be effectively unbounded. Our artifact [1] includes and documents the tuning ranges for all examined compiler flags. Each OpenTuner run performs 160 iterations. Even with this limited number of iterations, the result of OpenTuner in several benchmarks outperforms `-O3`. On RISC Zero, 18 out of 58 programs outperform `-O3` in terms of cycle count reduction (on average by 6.3%), while on RISC Zero, 20 programs outperform `-O3` (on average 14.6% cycle count reduction).

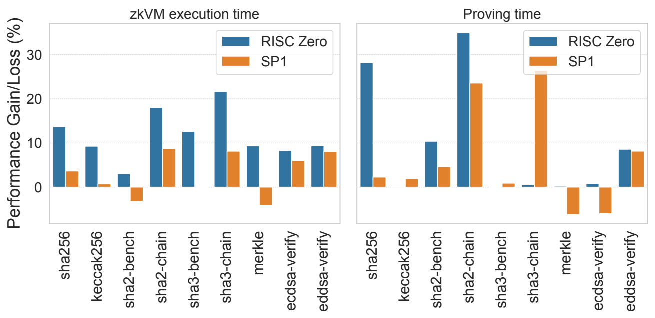
We further study the most common sub-sequences of passes appearing in the best and worst configurations across all benchmarks and zkVMs. Some findings align with the results of RQ1 (Section 4.1). For example, the `inline` pass appears in 573 out of 580 best-performing sequences, where 580 is the total number of top-5 sequences considered (2 zkVMs × 58 programs × 5 sequences each). In contrast, the `licm` pass is present in 385 of the worst-performing ones. This indicates that `inline` tends to be beneficial, and `licm` often detrimental, even when combined with other passes.

However, we also observed more interesting behaviors: `inline` appears in 122 of the worst-performing sequences, often when followed by `licm` (23 out of 122). In contrast, the combination where `licm` follows `inline` occurs 50 times in

[1] https://github.com/thomasgassmann/zkvm-compiler-optimizations



(a) NPB benchmark suite



(b) Crypto benchmark suite

Figure 6. The speedup over `-O3` (with LTO) achieved by tuning optimization passes for each program in the NPB and Crypto benchmark suites across both zkVMs.

best-performing sequences. Further analysis in Section 5 explores the root causes behind these context-sensitive effects.

Autotuning specific benchmark suites. In the previous experiment, each benchmark was tuned individually using cycle count as the fitness function. Yet, how does this translate to zkVM execution and proving time? To answer this, we autotune (1) the NPB benchmark [41] suite and (2) cryptography benchmarks on both zkVMs, running OpenTuner for a longer time frame (1,600 iterations instead of 160). We select the NPB suite for its large programs, which are more likely to benefit from optimization, and cryptography benchmarks as they best represent typical zkVM workloads.

Figure 6 shows the performance improvement of each program in the selected benchmark suites relative to `-O3` with link-time optimizations (LTO). For the NPB benchmark suite (Figure 6a), the autotuned configurations achieve an average performance increase by 17% on SP1 in terms of both zkVM execution time and proving time. On RISC Zero, autotuning improves by 19% the execution time, and by 17% the proving time. The `npb-sp` program stands out, as the result of the OpenTuner achieves over 2× speedup on both zkVMs for both metrics.

Moving to Figure 6b, autotuning yields smaller, yet meaningful, gains for the cryptography benchmarks. On average, execution improves by 11.6% on RISC Zero and 10.4% on SP1, while proving time improves by 3.5% and 6.8%, respectively. The smaller gains are expected, as both zkVMs are already heavily optimized for cryptographic workloads through the use of *precompiles*. These are specialized, built-in circuits that implement certain expensive operations (e.g., elliptic-curve signature verification or hashing) directly in hardware-equivalent logic, instead of executing them step-by-step as regular instructions. This allows these operations to be proven far more efficiently, often in constant time.

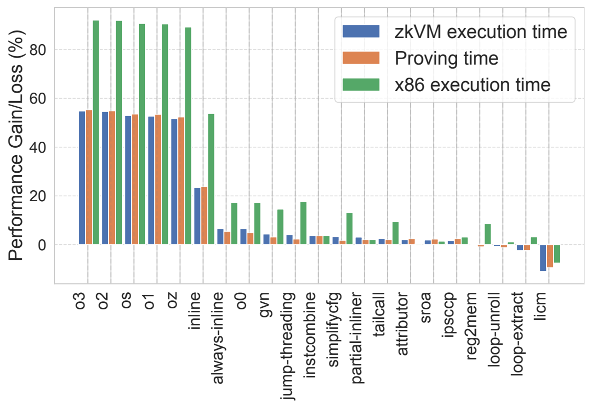


Figure 7. The average impact of each optimization on zkVM and x86 performance, measured across all benchmarks. Passes with an average effect below 2% are omitted.

Nevertheless, even for benchmarks that make use of such precompiles, such as `ecdsa-verify`, `eddsa-verify`, and `keccak256`, autotuning can still deliver performance improvements that exceed 10%.

The findings confirm the viability of autotuning on zkVMs, which has practical implications for optimizing performance-critical programs on zkVMs (see Section 6.2).

Security-critical bug in SP1. As a by-product, our autotuning efforts uncover a security-critical bug in SP1. In particular, while autotuning a specific program from the Crypto benchmark suite, we run into a certain sequence of passes that leads to 59% reduction in cycle count compared to `-O3` (w/ LTO)! This, however, turns out to be a bug in SP1 that causes the zkVM to *silently* abort execution in mid-run while still producing a proof that passes verification. Such silent failures represent a serious vulnerability, as they allow incorrect program results to be proven as correct. We privately report this silent-failure case to the SP1 development team, who confirm it as a bug and then patch it.

4.3 RQ3: Comparison to Traditional CPUs

We compare the impact of our studied LLVM passes and flags on zkVMs versus a traditional architecture, that is, x86.

Figure 7 shows the average impact of each pass on zkVM and x86 performance, measured across all benchmarks, relative to the baseline where no optimizations are applied, including Rust's MIR optimizations. For brevity, we exclude

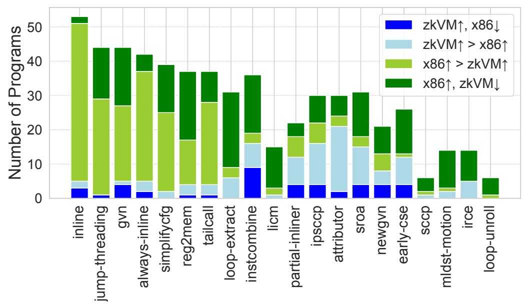


Figure 8. Number of programs where each pass exhibits: (1) a performance gain on x86 but a loss on RISC Zero, (2) a gain on both x86 and RISC Zero with >5% higher gain on x86, (3) a gain on both with >5% higher gain on RISC Zero, or (4) a gain on RISC Zero but a loss on x86.

passes with an average impact on zkVM execution time, proving time, and x86 execution time that is below 2%. For the majority of passes, the direction of their impact (positive or negative) is similar on both zkVMs and x86. For example, `inline` is the most beneficial pass on both platforms, while `licm` is the most detrimental pass when applied in isolation. The latter occurs because `licm` relies on subsequent passes to eliminate redundant `getelementptr` operations, which cause significant performance regressions (Section 5).

However, the magnitude of the effect is significantly larger on x86. This could be explained by the fact that many LLVM optimizations fail to show their full potential on zkVMs. This is mainly due to a mismatch in the cost model or the underlying heuristics of LLVM, which rely on assumptions that hold only on the traditional architectures. We provide further evidence on this in Section 5.

Figure 8 illustrates passes with divergent effects on x86 vs. RISC Zero execution. While most passes benefit both platforms, the performance improvement is generally more pronounced on x86. This exemplified by passes such as `inline`, `simplifycfg`, and `jump-threading`. Some passes, such as `reg2mem` and `loop-extract`, tend to benefit x86 but degrade performance on RISC Zero. In contrast, passes like `ipsccp` and `attributor` show stronger positive effects on RISC Zero. The corresponding plots for SP1 and proving time follow similar trends and are omitted for brevity.

5 Performance Impact Analysis

We examine the results of our study (Section 4) to distill the key factors that influence zkVM performance and explain why certain optimization passes perform well or poorly.

5.1 Cost Components

According to RQ1 (Section 4.1), `inline` is the most beneficial pass for zkVMs, while `licm` is the most detrimental pass for zkVM performance. To understand *why*, we analyze the underlying cost components. Beyond *cycle count*, which directly reflects execution cost (Section 4.1), we examine the *dynamic instruction count*, i.e., the number of executed instructions. Our insight is that since zkVM's model assigns almost a uniform cost for each instruction, executing fewer instructions could potentially lead to performance gains. Finally, for RISC Zero (the corresponding metric is not available in SP1), we consider *paging cycles*, which measure the cycles required to move data between the guest program's memory and the zkVM's storage system. Page-ins and page-outs introduce high overhead in zkVMs. For example, on RISC Zero a paging operation costs roughly 1,130 cycles [1].

Table 2. Monotonic and linear relationships between zkVM cost metrics and performance, reported as average Kendall's and Pearson's coefficients. Correlations are computed per benchmark over optimization variants.

Performance metric	Cost Metric	SP1 Kendall's τ	SP1 Pearson	RISC Zero Kendall's τ	RISC Zero Pearson
zkVM exec time	Executed instr.	0.87	0.99	0.58	0.97
	Paging cycles	N/A	N/A	0.38	0.88
	Total cycles	0.87	0.99	0.59	0.97
Proving time	Executed instr.	0.49	0.9	0.5	0.95
	Paging cycles	N/A	N/A	0.33	0.89
	Total cycles	0.49	0.89	0.48	0.95
Paging cycles	Executed instr.	N/A	N/A	0.43	0.9

Figure 9 illustrates the impact of the two most beneficial passes (`inline` and `always-inline`), and the two most detrimental passes (`loop-extract` and `licm`) on execution time and proving time in RISC Zero, relative to the baseline with no optimizations. For completeness, we also show the corresponding plots for `-O3` and `-O0`. The other passes and pipelines exhibit similar trends and can be found in our artifact. [2] The plots also report the effect of these passes on key zkVM cost components: cycle count, dynamic instruction count, and paging cycles. There is a clear trend: when a pass (e.g., `inline`) yields significant improvements in execution or proving time, it also leads to substantial reductions in the underlying cost metrics (e.g., `inline` on `polybench-floyd-warshall`). Similarly, performance regressions are mainly attributed to tremendous increases in either dynamic instruction count or paging cycles, or all of them (e.g., `licm` on `npb-lu`). When performance gains in execution or proving time are more modest, the corresponding reductions in cost metrics are also limited, or different metrics exhibit opposing effects, such as a decrease in executed instructions accompanied by an increase in paging cycles (e.g., `always-inline` on `factorial`).

Monotonicity and linearity. To assess whether these cost components can reliably guide compiler optimization decisions for zkVMs, we evaluate the monotonic relationship between dynamic instruction count, paging cycles, and zkVM performance. To this end, for each benchmark, we compute Kendall's τ correlation coefficient [31] between each cost

[2] https://github.com/thomasgassmann/zkvm-compiler-optimizations

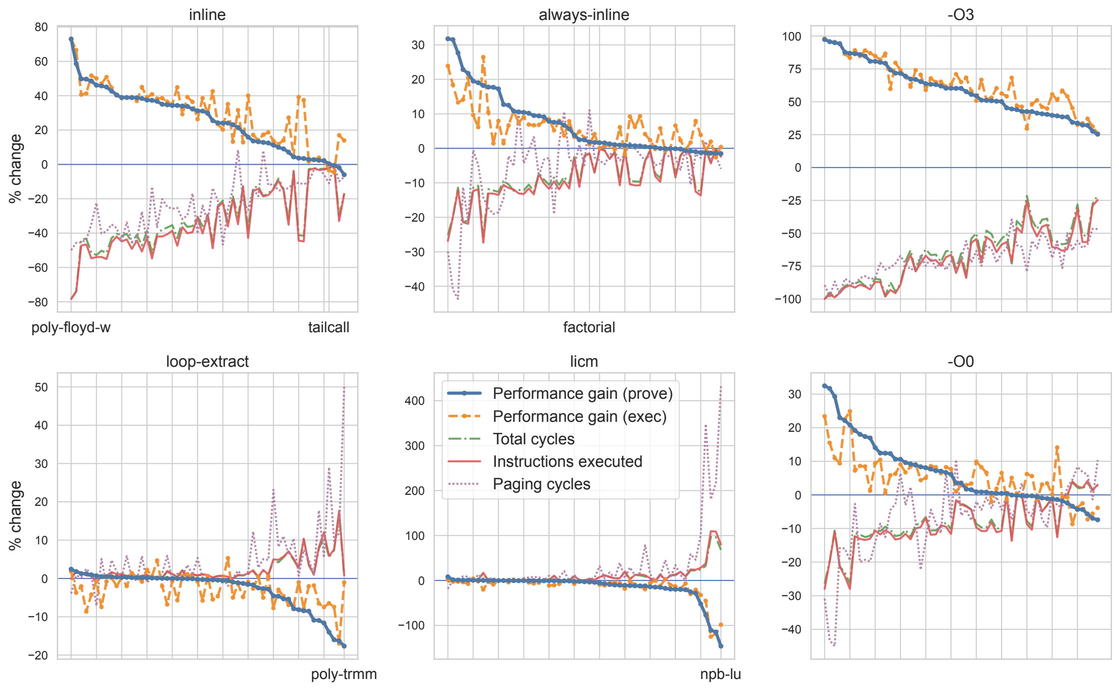


Figure 9. Impact of representative passes on RISC Zero performance and cost metrics. Each plot shows the percentage change relative to the baseline for proving time and execution time, along with changes in key zkVM cost components: total cycles, dynamic instruction count, and paging cycles. The x-axis represents benchmarks. For execution and proving time, higher values indicate performance improvements, whereas for the cost metrics, higher values correspond to increased overhead.

metric (dynamic instruction count, paging cycles, total cycles) and both zkVM execution time and proving time, across individual passes applied in isolation. We also report the Pearson correlation coefficient, which captures linear relationships. Table 2 summarizes the results.

Overall, dynamic instruction count and paging cycles exhibit moderate-to-strong positive correlations with both execution time and proving time across zkVMs. For example, a Kendall's τ value of 0.58 between dynamic instruction count and RISC Zero execution time indicates that, for a given benchmark, there is a 79% probability that an optimization increasing dynamic instruction count also increases execution time. This probability is computed by $(1 + \tau)/2$, where τ is the Kendall's coefficient. The high Pearson coefficients (> 0.9) further suggests that this relationship is linear.

For RISC Zero, dynamic instruction count shows stronger correlation with both execution time and proving time than paging cycles. While dynamic instruction count is the primary cost driver, the two metrics are complementary, as reflected by a Kendall's τ of 0.43 between them. In 28.5% of cases, the metrics disagree: an optimization may reduce instruction count while increasing paging overhead, or vice versa (e.g., Figure 9; `always-inline` on `factorial`).

```rust
const N: usize = ...;
let mut v = [0; N];
for k in 0..N {
  v[k] = 42;
}
```

(a) A simple loop of depth one.

```rust
const N: usize = ...;
let mut v = ...;
for k in 0..N {
 for j in 0..N {
  for i in 0..N {
   for l in 0..N {
    v[k][j][i][l] = 42;
... }
```

(b) Nested loops of depth four.

Figure 10. Simplified examples that cause significant performance regressions when using `licm`.

5.2 Case Studies

Based on our study's results (Section 4), we examine a selection of passes with interesting performance behaviors. We select these passes based on four criteria: (1) passes that cause considerable performance losses on zkVMs, (2) passes that provide significant performance gains on zkVMs, (3) passes that benefit x86 execution but degrade performance on zkVMs, and (4) passes that improve both x86 and zkVM performance, but with a substantially greater impact on x86. Our analysis aims to identify the root causes of these performance behaviors based on the main zkVM cost components (Section 5.1). Then, we derive a set of principles that can

```
1   pub fn work(x: u64) -> u64 {
2     let mut sum = x;
3     for j in 0..100 {
4       sum = sum.wrapping_mul(31).wrapping_add(j);
5     }
6     sum
7   }
8   fn main() {
9     let n = 1000;
10    for i in 0..n { let res = work(i as u64); }
11  }
```

Figure 11. Example Rust function that `inline` causes a performance regression for zkVMs.

```
pub fn matmul(mat: &[[f64; 5]; 5], vec: &[f64; 5]) {
  let mut res = [0.0; 5];
  for col in 0..5 {
    for row in 0..5 {
      res[row] += mat[col][row] * vec[col];
... }
```

Figure 12. Example Rust function that computes the matrix-vector product of a 5 × 5 matrix and a 5-dimensional vector.

guide potential LLVM improvements for better zkVM performance (Section 6.1).

Significant performance regressions on zkVMs. `licm` is the most detrimental pass for zkVM performance in RQ1 (Section 4.1). The effect is particularly severe on the `npb-lu` benchmark, where `licm` slows down proving time by 2.7× on RISC Zero and 1.7× on SP1, and execution by 1.7× on both. The root cause is a sharp increase in *paging* cycles: +444% on RISC Zero, +69% on SP1. Indeed, across all benchmarks, `licm` increases paging cycles by 32%.

Figure 10a illustrates a simplified loop extracted from `npb-lu` where `licm` incurs this overhead. This comes from the increased number of `getelementptr` instructions in the LLVM IR, each computing the address of an array element to store the constant 42 (line 4). These extra memory operations in turn cause significant paging pressure with an increase in page-ins and page-outs.

The pattern is also exemplified by the program in Figure 11 (extracted from the `tailcall` program). Here, inlining causes major regressions on both zkVM execution (0.8× speedup) and proving (0.45× speedup). The root cause is stack spilling: RISC-V has only 32-bit registers, so each 64-bit variable occupies two. After inlining the function `work` (lines 1 and 10), three `u64` variables (outer-loop index `i` on line 10, inner-loop index `j` on line 3, and `sum` on line 2) must coexist. This forces the compiler to spill both `j` and `sum` to the stack every inner-loop iteration. This doubles the `lw`/`sw` (load/store) instructions, which account for most of the 2× cycle increase. Indeed, the number of the executed instructions increases by 102%, while the number of paging cycles increases by 54%.

Principle 1 (P1): Optimizations that introduce a lot of paging pressure should be avoided, as it harms zkVMs. Examples of passes that can potentially increase paging pressure include `licm`, `inline`, and `reg2mem`. Since page-ins and page-outs are highly expensive in zkVMs, compilers targeting zkVMs must prioritize paging-aware cost models when transforming loops or spilling to memory.

To validate this principle, we manually increase the nesting of the loop of Figure 10a. Nested loops exacerbate the negative effects of paging pressure: with a nesting of depth 4 (Figure 10b), `licm` increases paging cycles and dynamic instruction count by 46% and 155%, respectively, compared to 7% and 25%, at a depth 2. The extra memory pressure stemming from `licm` is evident when loops iterate over arrays, as this leads to a higher number of load/store operations.

Significant improvements on zkVMs. According to RQ1 (Section 4.1), `inline` is the most beneficial pass for zkVMs. But is LLVM's default inlining optimal [52]? To find out, we raise the inlining threshold from 225 (default) to 4328 using the `-inline-threshold` option, allowing the compiler to inline more aggressively even larger functions. Notably, the value of 4328 comes from the OpenTuner's autotuning results on the NPB and crypto benchmarks (Section 4.2). Compared to `-O3` with the default threshold, the higher `-inline-threshold` improves execution by 6% on RISC Zero and 1% on SP1 on average, with some benchmarks showing dramatic gains, such as +44% on RISC Zero and +40% on SP1 for the `npb-bt` program. These performance gains are accompanied by similar decreases in both dynamic instruction count and paging cycles. In contrast, increasing the inline threshold degrades x86 performance on average by 1%.

Principle 2 (P2): Applying inlining selectively based on dynamic instruction count is beneficial for zkVMs. Our analysis in Figure 9 shows that reductions in executed instructions caused by `inline` often lead to substantial zkVM improvements, primarily by eliminating call overhead. However, as shown in Figure 11, inlining should be avoided if it causes spills to the stack, as load/store instructions increase dynamic instruction count and may trigger expensive memory paging (Figure 11). This suggests that compilers targeting zkVMs should: (1) adopt a zkVM-aware cost model for inlining rather than reusing heuristics from traditional CPUs, and (2) tune inlining thresholds per workload, especially for performance-critical code (Section 6.2).

To validate this principle, we manually inline selected functions in the `tailcall` benchmark by adding `#[inline(always)]` annotations, while excluding the function that triggers stack spills. This reduces dynamic instruction count by 5%, yielding a 2% faster execution and eliminating the previous regression.

Improvements on x86; degradations on zkVMs. On `polybench-durbin`, the `loop-unroll` pass improves x86 execution by 9% but degrades RISC Zero performance by 8.7%.

Table 3. Relative speedup and change in dynamic instruction count for 4× and 16× loop unrolling, compared to the non-unrolled assembly code.

Unrolling Factor	x86	SP1			RISC Zero		
		#instr.	prove	exec	#instr	prove	exec
4	+28.1%	-50%	+24.3%	+6.6%	-50%	+51.4%	+39.5%
16	+31.5%	-63%	+26.4%	+1.5%	-62%	+51.6%	+52.7%

Figure 12 shows an oversimplified version of this benchmark where `loop-unroll` produces this divergence between x86 and zkVM performance. Overall, x86 sees gains of 9% when loop unrolling this example program compared to the unoptimized baseline, whereas both zkVMs consistently suffer slowdowns of 3%–10% in both execution and proving time. The regressions are caused by a 2% increase in executed instructions and 17% increase in paging cycles on both zkVMs.

When used with `-O3` and a high `unroll-threshold` (aggressive unrolling), loop unrolling becomes beneficial compared with `-O3` and no loop unrolling. Specifically, on RISC Zero, execution improves by 16% and proving by 3.7%; on SP1, by 0.5% and 1.7%. In both zkVMs, these performance gains are attributed to by 4–5% decrease in cycle count (explained by a 5% decrease in dynamic instruction count). However, these gains are still modest compared to x86's 32× speedup improvement. This trend also appears in `polybench-gemm`, which is also a matrix multiplication benchmark. By increasing the unrolling threshold, we save around 43–44% in terms of cycles (compared to `-O3`), by executing 44% fewer instructions. This leads to a 11.5% improvement in execution time on RISC Zero and 2% improvement on SP1.

Principle 3 (P3): Performance gains are observed only when loop unrolling leads to a reduction in the executed instructions. Since zkVMs cannot exploit instruction-level parallelism, out-of-order, or superscalar execution, performance gains when unrolling loops depend entirely on reducing instruction count. Compilers targeting zkVMs should prioritize instruction reduction over traditional hardware assumptions.

To validate this principle, we manually unroll loops in RISC-V assembly of program shown in Figure 12, avoiding interference from other passes or vectorization. Table 3 reports speedups for 4× and 16× loop unrolls. This manual effort yields substantial performance gains for both zkVMs, with RISC Zero even surpassing the gains seen on x86. These improvements come mainly from executing fewer loop bookkeeping instructions, such as counter increments, comparisons, and branch jumps. As dynamic instruction count decreases, paging cycles also decrease, by about 40% on average.

Substantial improvement on x86; minimal effect on zkVMs. On `polybench-nussinov`, `simplifycfg` improves x86 by 23.6%, but SP1 execution by only 2.6%. Figure 13 shows a minimized function from this benchmark that computes the absolute value of an integer using a branch. `simplifycfg` replaces the branch with equivalent arithmetic operations (Figures 13b and 13c). When tested with an array of integers chosen at random, the optimized version runs 2.2× faster on x86, presumably thanks to less branch misprediction overhead. However, on zkVMs, the optimized code increases cycle count by 17.7% on RISC Zero and 7.6% on SP1. These are primarily explained by increases in the number of executed instructions (8%) and paging cycles (17%). As a result, proving time degrades by 18.1% and 7% on RISC Zero and SP1, respectively, while execution time worsens by 8.5% and 7.7%.

```rust
fn abs_i32_branchy(x: i32) -> i32 {
  if x < 0 { -x } else { x }
}
```

(a) Rust function computing $|x|$ using a branch.

```
bltz a0, .LBB52_2
ret
.LBB52_2:
neg a0, a0
ret
```

(b) RISC-V before applying `simplifycfg`.

```
srai a1, a0, 31
xor a0, a0, a1
sub a0, a0, a1
ret
```

(c) RISC-V after applying `simplifycfg`.

Figure 13. An example Rust function including its assembly output before and after applying `simplifycfg`.

Principle 4 (P4): Branches should not always be eliminated, as branch elimination may increase the number of executed instructions by requiring both paths to be evaluated . Compilers targeting zkVMs should be more conservative when removing branches, for two reasons. First, branches are relatively cheap and incur no misprediction penalty compared to traditional CPUs. Second, predicated execution increases proving cost, since both paths must be proven and executed.

To validate this principle, we modify the unoptimized binary of `polybench-nussinov` by replacing the implementation of the `max` function with a branchless version implemented with right-shifts and bitwise `AND` operations. While this change yields a 4% performance improvement on x86, it degrades performance on RISC Zero by 4.4% and on SP1 by 0.5%. This regression is caused by a 3% increase in dynamic instruction count on both zkVMs.

6 Implications

We demonstrate the practical value of our study's results by refining existing LLVM passes to improve zkVM performance and then extract several key lessons from our findings.

6.1 zkVM-Specific Optimizations

Existing LLVM optimization levels already boost zkVM performance (Section 4.2), but our root cause analysis (Section 5) reveals significant untapped potential. Building on these insights, we propose a *non-exhaustive* set of lightweight backend and pass modifications across three dimensions to further unlock LLVM's potential for zkVMs.

Change set 1: Adopting a zkVM-specific cost model. The default RISC-V cost model in LLVM is not zkVM-aware, yet instruction costs on zkVMs differ greatly from traditional CPUs. For example, multiplication costs the same as addition, and division is not expensive. Memory access is also far more expensive when it triggers page-ins or page-outs (see *Principle 1*, Section 5). We address these by updating the `RISCVTTIImpl` and `RISCVTargetLowering` classes, which provide target-specific cost models to LLVM's middle-end so that passes can make better decisions for this architecture.

Change set 2: Updating defaults and heuristics. Many LLVM passes employ heuristics that are tuned for traditional architectures. This tuning may not be suitable for zkVMs. Adjusting these for zkVM-specific instruction costs, such as applying function inlining (*Principle 2*) or loop unrolling (*Principle 3*) only when it reduces instructions count can improve the performance of the generated code.

We increase the default inlining threshold and adjust parameters such as `inline-call-penalty` accordingly. We also update `simplifycfg` to make branch elimination more conservative and avoiding evaluation of both sides of a predicated operation (*Principle 4*).

Change set 3: Disabling irrelevant passes. Our RQ1 analysis (Section 4.1) shows that some passes offer little or no benefit on zkVMs, mainly because they rely on hardware features that are unsupported in zkVMs. For example, `speculative-execution` hoists side-effect-free instructions to reduce branch-misprediction latency on out-of-order, superscalar CPUs. This advantage is irrelevant to zkVMs, which execute RISC-V code strictly in order inside an arithmetic circuit. Similarly, we also disable `loop-data-prefetch`, and `hot-cold-splitting`, as they provide no measurable gain.

All these targeted LLVM changes took *only* a few hours to implement and required fewer than 100 lines of code.

Results. We evaluate `-O3` in our modified LLVM against `-O3` in the original toolchain. Figure 14 shows that our LLVM changes improve zkVM performance across a wide range of benchmarks, with execution-time gains exceeding 10% in many cases and reaching up to 45% on RISC Zero for `fibonacci`. This improvement directly validates our cost model and core principle (Section 5.1): reducing dynamic instruction count translates to proportional performance gains. The `fibonacci` benchmark experiences a 50% reduction in dynamic instruction count, which corresponds to the 45% execution time improvement on RISC Zero. Specifically, our division cost-model update enables LLVM to select a *single* `remu` instruction instead of a *series* of bitwise operations for remainder calculation.

The modified LLVM outperforms vanilla `-O3` on execution time in 39/58 benchmarks on RISC Zero (avg. +4.6%) and 19/58 on SP1 (avg. +1%), with only two and one regressions, respectively. On SP1, proving time improves for 27/58 programs by more than 1%, with only three regressions. On

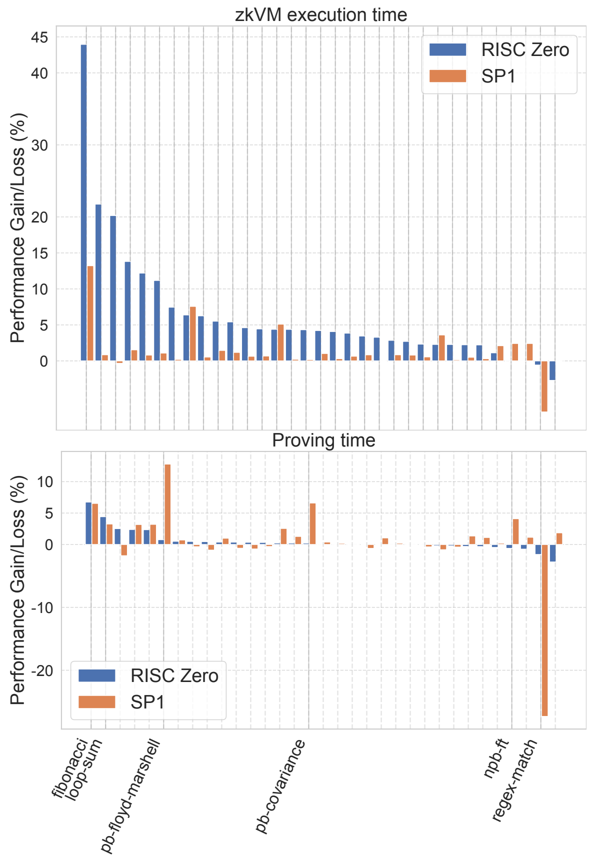


Figure 14. Changes in zkVM execution performance `-O3` with our modified LLVM compared to `-O3` with the original LLVM. Programs with effects below 2% are omitted.

RISC Zero, most proving-time effects are small, with 8 programs improving by more than 1% and 4 programs showing slight regressions. One outlier is `regex-match` on SP1, where our modifications lead to a 27.3% regression in proving time. This is caused by our changes in the `inline`'s heuristics, which make SP1 produce 20 proof shards instead of 16. Proof sharding divides the program into smaller proving units; more shards may increase proving time due to additional aggregation overhead.

Notably, our performance improvements are mainly explained by dynamic instruction count reduction: our modifications reduce the number of the executed instructions by 2.6% on average across both zkVMs, with only three benchmarks showing slight increases (max +0.5%).

6.2 Lessons Learned and Discussion

zkVM users can rely on the compiler to optimize their programs. Our results show that zkVM users can already benefit from LLVM's existing optimization levels (e.g., `-O3`), which improve both execution and proving time by over 40% (RQ2, Figure 5). However…

…the magnitude of these gains is smaller compared to what similar optimizations achieve on traditional architectures. Our fine-grained pass-level analysis (Section 4.1) reveals that many passes still rely on hardware-centric assumptions and heuristics that harm zkVM performance (e.g., branch elimination, Section 5). By refining just a few of these passes to be zkVM-aware, we already surpass LLVM's default `-O3` (Figure 14). This proof-of-concept highlights that even small, yet targeted zkVM-specific compiler changes can deliver substantial speedups and motivate the development of a dedicated zkVM-aware LLVM backend.

Section 5 covers only a few passes (e.g., `licm`, `inline`). Many others could unlock further optimization opportunities. For example, `instcombine` improves half the benchmarks but hurts the rest (Figure 4). Performing a similar root-cause analysis on this pass could reveal which rewrites of `instcombine` to keep and which to disable.

Autotuning performance-critical software. As part of RQ2 (Section 4.2), we also show the viability of program autotuning on zkVMs. This has direct, tangible value in real-world settings. For example, in the Ethproofs ecosystem [21], where different zkVMs race to prove Ethereum blocks in real-time (i.e., in less than 12-seconds – Ethereum's block time). Despite significant progress, real-time proving still requires substantial GPUs, and much research and engineering work focuses on achieving even minor improvements. Exploring autotuning for prover workloads on real Ethereum blocks is thus a promising and economically impactful direction.

Another takeaway from our autotuning experiment is that it can yield significant gains even for programs using precompiles, with improvements of up to 10%. This suggests that compiler-level tuning can complement zkVMs' built-in accelerations by further reducing proving costs.

Profile-guided optimization. High-quality profiling data is critical for generating efficient code. For example, static profile-guided optimization could significantly improve inlining decisions and reduce instruction count. Profiling data could also guide the design of future zkVM-specific passes. For example, a page coalescing pass could place hot variables within as few 1 KB pages as possible, minimizing expensive page-ins and page-outs on zkVMs, which we identify in Section 5 as a major bottleneck.

Combined cost metrics. Our analysis in Section 5.1 establishes that zkVM performance is primarily driven by dynamic instruction count. Paging cycles provide complementary information. This motivates the use of combined cost models that integrate multiple components into a single performance predictor. While this two-factor model explains the majority of performance variation (Table 2), some effects remain unexplained, such as SP1's proof sharding overhead in Section 6.1. Future work could refine the model by incorporating more zkVM-specific factors (e.g., use of `precompiles`), or learning workload-specific weights (e.g., memory-heavy vs. computation-heavy programs).

Use of superoptimization. Since zkVM execution and proving are orders of magnitude more expensive than native execution, it is often worth trading longer compilation time for faster proving. Superoptimization is a technique that allows compilers to do so by finding the optimal sequence of instructions for implementing a given computation. Superoptimizers that operate on LLVM IR (e.g., Souper [47]) could be adapted to zkVMs. Incorporating a zkVM-specific cost model would make them a powerful tool for generating highly efficient zkVM guest code.

7 Related Work

zkVMs. Ben-Sasson et al. [12] introduce the first system to prove executions of a von Neumann machine, albeit on a *bounded* machine with a predetermined cycle count. The field has since moved to *unbounded* proving, e.g., Cairo [26] which leverages STARKs [10] to scale. Recent efforts improve zkVM performance via better circuit generation and prover backends as well as more efficient proof systems [6, 13, 26, 34, 49]. Our work complements these improvements by studying how compiler optimizations interact with zkVM cost models, and outline directions to reduce proving overhead through traditional (or novel) optimizations.

Benchmarking ZKPs. Benchmarking ZKPs is crucial for identifying bottlenecks and guiding improvements. Ernstberger et al. [19] introduce ZK-BENCH, a framework for benchmarking ZK DSLs [9]. Chaliasos et al. [17] provide the first large-scale systematic study of ZK-EVMs, highlighting design trade-offs and bottlenecks. Industry efforts benchmark zkVMs to compare capabilities and performance [2, 30, 36]. We build on these benchmarks by reusing selected industry workloads and complement them with a broad and diverse Rust/C suite.

Autotuning & Superoptimization. Ren et al. [46] develops BinTuner to investigate the impact of compiler optimizations on binary code difference. BinTuner is based on OpenTuner [5], a framework for building domain-specific autotuners using genetic algorithms. In this work, we leverage OpenTuner to show that autotuning is a viable method to improve the proving performance of zkVMs for a given program. A complementary line of work is *superoptimization* [42, 47, 48], which searches for optimal instruction sequences for a given (typically loop-free) fragment. Exploring superoptimization for zkVMs is a promising future work, and our study provides valuable insights to guide it (Section 6.2).

8 Conclusion

We presented the first systematic study of how compiler optimizations affect two production RISC-V zkVMs. Most passes help, yet gains are modest and a few are harmful (e.g., `licm`). Autotuning yields up to 2.2× proving speedups beyond `-O3`. We established a strong correlation between dynamic instruction count and zkVM performance and, based on this insight, prototype zkVM-aware LLVM changes that further improve zkVM performance.

Future work can extend this study to additional zkVMs and proof systems, unifying cost metrics, and exploring superoptimizations for zkVMs. These compiler-centric directions can push zkVM proving performance beyond what proof-system research and engineering alone can deliver.

References

[1] 2025. RISC Zero Guest Optimization Guide. https://dev.risczero.com/api/zkvm/optimization.

[2] a16z crypto. 2024. a16z crypto zkVM benchmarks. https://github.com/a16z/zkvm-benchmarks.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[4] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2023. Ligero: lightweight sublinear arguments without a trusted setup. *Des. Codes Cryptogr.* 91, 11 (2023), 3379–3424. doi:10.1007/S10623-023-01222-8

[5] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Edmonton, Canada. https://commit.csail.mit.edu/papers/2014/ansel-pact14-opentuner.pdf

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

[7] Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Yan Ji, Jonas Lindstrøm, Deepak Maram, Ben Riva, Arnab Roy, Mahdi Sedaghat, and Joy Wang. 2024. zkLogin: Privacy-Preserving Blockchain Authentication with Existing Credentials. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM, 3182–3196. doi:10.1145/3658644.3690356

[8] Barry Whitehat. 2018. Roll Up Token. https://github.com/barryWhiteHat/roll_up_token Accessed: 2025-06-01.

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

[10] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.* (2018), 46. http://eprint.iacr.org/2018/046

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

[12] 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

[13] Jeremy Bruestle and Paul Gafni. 2023. RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity. (2023).

[14] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 41–42. doi:10.1145/3185768.3185771

[15] Vitalik Buterin. 2025. Simple L1: Make a successful L1 by only doing the bare essentials. https://vitalik.eth.limo/general/2025/05/03/simplel1.html

[16] Stefanos Chaliasos, Jens Ernstberger, David Theodore, David Wong, Mohammad Jahanara, and Benjamin Livshits. 2024. SoK: What don't we know? Understanding Security Vulnerabilities in SNARKs. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/chaliasos

[17] Stefanos Chaliasos, Itamar Reif, Adrià Torralba-Agell, Jens Ernstberger, Assimakis Kattis, and Benjamin Livshits. 2024. Analyzing and Benchmarking ZK-Rollups. In *6th Conference on Advances in Financial Technologies, AFT 2024, September 23-25, 2024, Vienna, Austria (LIPIcs, Vol. 316)*, Rainer Böhme and Lucianna Kiffer (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:24. doi:10.4230/LIPICS.AFT.2024.6

[18] NASA Advanced Supercomputing (NAS) Division. 2024. NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html

[19] Jens Ernstberger, Stefanos Chaliasos, George Kadianakis, Sebastian Steinhorst, Philipp Jovanovic, Arthur Gervais, Benjamin Livshits, and Michele Orrù. 2024. zk-Bench: A Toolset for Comparative Evaluation and Performance Benchmarking of SNARKs. In *Security and Cryptography for Networks - 14th International Conference, SCN 2024, Amalfi, Italy, September 11-13, 2024, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14973)*, Clemente Galdi and Duong Hieu Phan (Eds.). Springer, 46–72. doi:10.1007/978-3-031-71070-4_3

[20] Jens Ernstberger, Stefanos Chaliasos, Liyi Zhou, Philipp Jovanovic, and Arthur Gervais. 2024. Do You Need a Zero Knowledge Proof? *IACR Cryptol. ePrint Arch.* (2024), 50. https://eprint.iacr.org/2024/050

[21] Ethereum Foundation. 2025. Ethproofs. https://ethproofs.org/ Accessed: 2025-08-12.

[22] Joseph Rafael Ferrer. 2020. polybench-rs. https://github.com/JRF63/polybench-rs.

[23] Agner Fog. 2022. *Instruction Tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*. Technical University of Denmark. https://www.agner.org/optimize/instruction_tables.pdf Last updated: 2022-11-04.

[24] Matteo Frigo and Abhi Shelat. 2024. Anonymous credentials from ECDSA. *IACR Cryptol. ePrint Arch.* (2024), 2010. https://eprint.iacr.org/2024/2010

[25] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. *IACR Cryptol. ePrint Arch.* (2019), 953. https://eprint.iacr.org/2019/953

[26] Lior Goldberg, Shahar Papini, and Michael Riabzev. 2021. Cairo - a Turing-complete STARK-friendly CPU architecture. *IACR Cryptol. ePrint Arch.* 2021 (2021), 1063. https://api.semanticscholar.org/CorpusID:237263398

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

[28] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Proceedings, Part II, of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9666*. Springer-Verlag, Berlin, Heidelberg, 305–326.

[29] John Guibas. 2024. SP1 Benchmarks. https://blog.succinct.xyz/sp1-benchmarks-8-6-24/ Accessed: 2025-08-15.

[30] Aligned Inc. 2024. zkVMs benchmarks. https://github.com/yetanotherco/zkvm_benchmarks.

[31] M. G. KENDALL. 1938. A new measure of rank correlation. *Biometrika* 30, 1-2 (06 1938), 81–93. doi:10.1093/biomet/30.1-2.81 arXiv:https://academic.oup.com/biomet/article-pdf/30/1-2/81/423380/30-1-2-81.pdf

[32] Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla. 2022. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 13510)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, 359–388. doi:10.1007/978-3-031-15985-5_13

[33] Succinct Labs. 2024. Reth Succinct Processor (RSP). https://github.com/succinctlabs/rsp.

[34] Succinct Labs. 2024. SP1 Technical Whitepaper. (2024). Whitepaper.

[35] Succinct Labs. 2024. Upgrading Blobstream to SP1. https://blog.succinct.xyz/celestia-sp1. https://blog.succinct.xyz/celestia-sp1 Accessed: 2025-07-15.

[36] Succinct Labs. 2024. zkvm-perf. https://github.com/succinctlabs/zkvm-perf.

[37] Succinct Labs. 2025. SP1 GitHub. https://github.com/succinctlabs/sp1. GitHub repository.

[38] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. doi:10.1109/CGO.2004.1281665

[39] Ryan Lavin, Xuekai Liu, Hardhik Mohanty, Logan Norman, Giovanni Zaarour, and Bhaskar Krishnamachari. 2024. A Survey on the Applications of Zero-Knowledge Proofs. *CoRR* abs/2408.00243 (2024). doi:10.48550/ARXIV.2408.00243 arXiv:2408.00243

[40] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 2023. GZKP: A GPU Accelerated Zero-Knowledge Proof System. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 340–353. doi:10.1145/3575693.3575711

[41] Eduardo M. Martins, Leonardo G. Faé, Renato B. Hoffmann, Lucas S. Bianchessi, and Dalvan Griebler. 2025. NPB-Rust: NAS Parallel Benchmarks in Rust. arXiv:2502.15536 [cs.DC] https://arxiv.org/abs/2502.15536

[42] Henry Massalin. 1987. Superoptimizer: a look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (Palo Alto, California, USA) *(ASPLOS II)*. Association for Computing Machinery, New York, NY, USA, 122–126. doi:10.1145/36206.36194

[43] Mashiat Mutmainnah. 2024. How to leverage RISC Zero's zkVM to scale Bitcoin. https://risczero.com/blog/how-to-leverage-risc-zeros-zkvm-to-scale-bitcoin

[44] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 238–252. doi:10.1109/SP.2013.47

[45] Louis-Noel Pouchet. 2010. PolyBench/C - the Polyhedral Benchmark suite. https://www.cs.colostate.edu/~pouchet/software/polybench/.

[46] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 142–157.

[47] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2018. Souper: A Synthesizing Superoptimizer. arXiv:1711.04422 [cs.PL] https://arxiv.org/abs/1711.04422

[48] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2012. Stochastic Superoptimization. arXiv:1211.0557 [cs.PF] https://arxiv.org/abs/1211.0557

[49] Srinath Setty, Justin Thaler, and Riad Wahby. 2023. Unlocking the lookup singularity with Lasso. Cryptology ePrint Archive, Paper 2023/1216. https://eprint.iacr.org/2023/1216

[50] Succinct Labs. 2025. SP1 Documentation: Hardware Acceleration. https://docs.succinct.xyz/docs/sp1/generating-proofs/hardware-acceleration. Accessed: 2025-08-12.

[51] Justin Thaler et al. 2022. Proofs, arguments, and zero-knowledge. *Foundations and Trends® in Privacy and Security* 4, 2–4 (2022).

[52] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 977–989. doi:10.1145/3503222.3507744

[53] Morgan Thomas, Mamy Ratsimbazafy, Marcin Bugaj, Lewis Revill, Carlo Modica, Sebastian Schmidt, Ventali Tan, Daniel Lubarov, Max Gillett, and Wei Dai. 2025. Valida ISA Spec, version 1.0: A zk-Optimized Instruction Set Architecture. *arXiv preprint arXiv:2505.08114* (2025).

[54] Linda Torczon and Keith Cooper. 2007. *Engineering A Compiler* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[55] Verified zkEVM Initiative. 2025. Verified zkEVM. https://verified-zkevm.org/

[56] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2016. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1.* Technical Report UCB/EECS-2016-118. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

[57] RISC Zero. 2025. RISC Zero - Recursive Proving. https://dev.risczero.com/api/recursion.

[58] RISC Zero. 2025. RISC Zero GitHub. https://github.com/risc0/risc0. GitHub repository.

A Differences between zkVMs and Native Execution

The following considers two popular zkVMs (RISC Zero [13] and SP1 [34]) and summarizes the key differences compared to traditional architectures.

Instruction latency. On traditional architectures, instruction latency can vary significantly depending on the instruction type, cache hits/misses, and other micro-architectural effects [23]. In zkVMs, however, most instructions have a uniform cost [1], with most instructions taking exactly the same number of cycles.

Memory access. Memory access is relatively cheaper on zkVMs. For example, on RISC Zero, memory access costs only one cycle if the page is already paged in. Page-ins and page-outs on RISC Zero are, however, more expensive, taking around 1130 cycles on average [1]. zkVMs also do not have any caches making memory access always synchronous.

Control flow. zkVMs do not have branch prediction or instruction caches, which means that control flow instructions can be cheaper on zkVMs, as there is no penalty for mispredicted branches.

Out-of-order execution. zkVMs do not have out-of-order execution. They also do not have super-scalar execution,

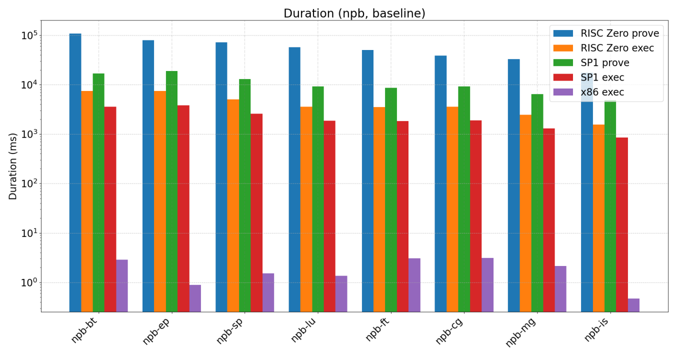


Figure 15. Median duration of zkVM operations compared to execution time on x86 for the NPB benchmark suite. All binaries were compiled with no optimizations.

instruction-level parallelism, or pipelining, which means that all instructions are executed sequentially.

Floating-point operations. zkVMs do not support native floating-point operations, which means that all floating-point operations must be emulated. This can lead to significant overhead, especially for complex floating-point operations.

Multithreading. zkVM execution is always single-threaded. There is no parallelism in the execution.

Precompiles. zkVMs provide precompiles for expensive operations such as cryptographic hashes or elliptic-curve arithmetic. Precompiles can significantly reduce the cost of these operations compared to implementing them in the guest program.

zkVM execution and proving are order of magnitude slower than native execution. Figure 15 illustrates this by considering the time spent on zkVM execution, proof generation, and native execution for every program included in the NPB benchmark suite [41].

B Benchmark Suite Details

In the following, we give a brief summary of the benchmark programs.

PolyBench. PolyBench is a benchmark suite of 30 numerical computations from domains like linear algebra, image processing, physics simulations, dynamic programming, statistics, and more. Originally written in C [45], we use the Rust port [22] of the benchmark suite. The inputs of the benchmarks have been reduced to fit within the constraints of the zkVMs.

NPB. The NAS Parallel Benchmarks (NPB) are a set of eight benchmarks designed to evaluate the performance of parallel supercomputers originally developed by NASA's Advanced Supercomputing Division [18]. Martins et al. [41] port the benchmarks to Rust and provide a sequential version of the benchmarks. We use the sequential version of the Rust port for our evaluation. Similar to PolyBench, the input sizes have been reduced to fit within the constraints of the zkVMs.

SPEC CPU 2017. The SPEC CPU benchmark suite [14] is a widely used benchmark suite for evaluating the performance of computer systems. Due to the complexity of the benchmarks and the fact that they are written in C/C++, whose support is still limited in zkVMs, we only use a subset of three benchmarks: `605`, `619` and `631`. These three benchmarks were chosen based on their ease of integration.

a16z crypto zkVM benchmarks. Andreessen Horowitz (a16z) crypto has published a set of benchmarks [2] comparing the RISC Zero, SP1 and Jolt [6] zkVMs. We use the benchmarks from their repository, which consist of a set of cryptographic operations (`sha2`, `sha3`) as well as an allocation-heavy benchmark (`bigmem`).

Succinct Labs benchmarks. We also incorporate a subset of the zkVM benchmarks released by Succinct Labs [36]. The benchmarks we include consist of a set of cryptographic operations (`ecdsa-verify`, `eddsa-verify`, `keccak256`) as well as a Fibonacci sequence benchmark (`fibonacci`).

Reth Succinct Processor (RSP). We also adopt the Reth Succinct Processor (RSP) [33] benchmarks from Aligned Inc.'s benchmark suite [30]. RSP represents a common type of workload for zkVMs that generates zero-knowledge proofs for EVM (Ethereum Virtual Machine) block execution. We use block 20526624 as input for our benchmarks.[3]

Others. In addition to the above benchmarks, we also add some additional benchmarks: a benchmark training a neural network on the MNIST dataset (`zkvm-mnist`) downsampled to 7x7 images, a benchmark performing regex matching (`regex-match`), a benchmark performing a merkle tree inclusion proof (`merkle`), another SHA-256 benchmark (`sha256`) as well as three smaller benchmarks (`factorial`, `loop-sum` and `tailcall`).

Overall, the 58 benchmark programs that we have assembled span a diverse range of workloads and input characteristics, ranging from numerical computations to memory-intensive operations, cryptographic primitives commonly used in zkVMs, and real-world applications such as RSP. This variety ensures a thorough evaluation of compiler optimizations across all major zkVM use cases.

Benchmark Programs. Table 4 summarizes the benchmark programs used in this evaluation, including their source lines of code (SLOC) counts for Rust and C/C++. The SLOC counts of any libraries used by the benchmarks are *not* included in the table.

A list of the library versions is listed in Table 5. All programs also use the crates `sp1-zkvm` (version 4.1.1) and `risc0-zkvm` (version 1.2.3).

Table 4. Benchmark programs and their SLOC counts, split across two subtables with balanced rows.

Program	Libraries used	Precomp.	Rust	C/C++	Total
bigmem	None	No	32		32
ecdsa-verify	k256	Yes	16		16
eddsa-verify	ed25519_dalek	Yes	16		16
factorial	None	No	26		26
keccak256	sha3	Yes	13		13
loop-sum	None	No	29		29
merkle	rs_merkle	No	46		46
npb-bt	None	No	2915		2915
npb-cg	None	No	827		827
npb-ep	None	No	485		485
npb-ft	None	No	1151		1151
npb-is	None	No	815		815
npb-lu	None	No	2746		2746
npb-mg	None	No	2121		2121
npb-sp	None	No	2728		2728
polybench-2mm	None	No	113		113
polybench-3mm	None	No	148		148
polybench-adi	None	No	107		107
polybench-atax	None	No	78		78
polybench-bicg	None	No	80		80
polybench-cholesky	None	No	63		63
polybench-correlation	None	No	98		98
polybench-covariance	None	No	84		84
polybench-deriche	None	No	146		146
polybench-doitgen	None	No	83		83
polybench-durbin	None	No	67		67
polybench-fdtd-2d	None	No	93		93
polybench-floyd-warshall	None	No	59		59
polybench-gemm	None	No	90		90
polybench-gemver	None	No	116		116
polybench-gesummv	None	No	81	81	
polybench-gramschmidt	None	No	88	0	88
polybench-heat-3d	None	No	82	0	82
polybench-jacobi-1d	None	No	64	0	64
polybench-jacobi-2d	None	No	70	0	70
polybench-lu	None	No	64	0	64
polybench-ludcmp	None	No	103	0	103
polybench-mvt	None	No	79	0	79
polybench-nussinov	None	No	99	0	99
polybench-seidel-2d	None	No	70	0	70
polybench-symm	None	No	88	0	88
polybench-syr2k	None	No	85	0	85
polybench-syrk	None	No	81	0	81
polybench-trisolv	None	No	67	0	67
polybench-trmm	None	No	74	0	74
regex-match	regex	No	45	0	45
rsp	serde, rsp-client-executor, c-kzg, bytemuck_derive, bincode	Yes	11	0	11
sha2-bench	sha2	No	36	0	36
sha2-chain	sha2	No	44	0	44
sha3-bench	sha3	No	35	0	35
sha3-chain	sha3	No	44	0	44
sha256	None	No	163	0	163
spec-605	None	No	48	2159	2207
spec-619	None	No	53	203	256
spec-631	None	No	46	6496	6542
tailcall	None	No	56	0	56
zkvm-mnist	None	No	181	0	181

Table 5. Programs and their associated libraries and versions.

Program	Library	Version/Commit
PolyBench [22]	–	babdd97
NPB [41]	–	4bd879b
ecdsa-verify	k256	0.13.3
eddsa-verify	ed25519_dalek	2.1.1
merkle	rs_merkle	1.5.0
regex-match	regex	1.11.1
sha2-bench	sha2	0.10.8
sha2-chain	sha2	0.10.8
sha3-bench	sha3	0.10.8
sha3-chain	sha3	0.10.8
rsp	serde	1.0.204
	rsp-client-executor	249b34e (SP1)
	rsp-client-executor	4ceefdf (RISC Zero)
	c-kzg	1.0.3
	bytemuck_derive	1.8.0
	bincode	1.3.3

Table 6. Statistics of execution and proving times (in seconds) for each zkVM across all benchmarks, measured on the unoptimized baseline with no compiler passes applied.

zkVM	Execution				Proving			
	Min	Max	Mean	Median	Min	Max	Mean	Median
RISC Zero	0.04	157.70	4.51	0.34	0.53	2071.24	60.85	3.83
SP1	0.06	41.81	1.70	0.23	0.38	205.87	8.89	1.90

C Statistics of Baseline

Table 6 shows the statistics of execution and proving times (in seconds) for each zkVM across all benchmarks, measured on the unoptimized baseline with no compiler passes applied.

[3]The FAQ on RSP's GitHub page highlights block 20526624 as "a good small block to test". While adapting our benchmark setup to test a different block is relatively straightforward, running the benchmarks for all optimization profiles is time-consuming. The impact of compiler optimizations on RSP under consideration of different blocks is left as future work.