Revealing Semantic Gaps in Zero-Knowledge Virtual Machines via Obligation-Targeted Semantic Fuzzing

Anonymous Authors
Paper #XXX (Anonymous Submission)

***Abstract*—Zero-knowledge virtual machines (zkVMs) prove correct execution of programs and underpin systems securing billions of dollars in digital assets. Their security depends on faithful alignment between the ISA and the *constraint system* (polynomial equations over a finite field) that certifies an execution trace of the program. Any deviation is a *semantic gap* of two operationally distinct kinds: *underconstraint* bugs where the constraints admit an alternative witness encoding a forbidden execution (a soundness violation, exploitable by a malicious prover), and *divergence* bugs where the executor diverges from the ISA so that an honest prover cannot certify a legitimate execution or crashes (a completeness violation). Such gaps cluster at non-trivial corner cases (e.g., division by zero, signed overflow) that fuzzing without constraint-level guidance almost never reaches.**

Our key insight is that these gaps correspond to *constraint obligations* derivable from two fixed sources: the ISA specification and the proof system's algebraic structure. We specify each obligation as a triple $(\mu_c, \kappa_c, \iota_c)$ and build BEAK around it: a reinforcement-learning fuzzing framework in which the three components replace the three default-random loci of a generic fuzzer. The enrichment template μ_c constructs the loop's initial corpus, ensuring every obligation is exercised from the first iteration; the predicate κ_c defines the bandit's reward via obligation-signature novelty, replacing the traditional edge-coverage reward; and the injection template ι_c anchors witness injection at activation steps, replacing infeasible blind witness-cell injection. Two complementary detectors run on every iteration: differential register comparison catches divergence bugs, and ι_c-anchored witness injection catches underconstraint bugs invisible to output.

Evaluated across all leading production RV32IM zkVMs, on a curated benchmark of 37 known bugs spanning both underconstraint and divergence classes, BEAK detects 26/37 (70.3%) versus the prior state of the art's 5/37 (13.5%), a $5.2\times$ improvement, with detections including all of the prior work's. BEAK additionally discovers 14 previously unknown zero-day findings in production systems, all responsibly disclosed and patched.

I. INTRODUCTION

A *zero-knowledge proof* (ZKP) lets a prover convince a verifier that a computation produced a given output without revealing intermediate state, with a succinct proof the verifier checks far faster than recomputation [1]. A *zero-knowledge virtual machine* (zkVM) lifts this guarantee to general programs: instead of writing a custom proof circuit per application, the developer writes ordinary code targeting a standard instruction set (most production systems target RISC-V), and the zkVM proves the entire execution [2]–[7]. zkVMs underpin blockchain rollups, privacy-preserving applications, and off-chain computation platforms collectively securing billions of dollars in digital assets; SP1-based deployments alone account for over $5.5B in total value secured across 35+ customers [8]. A bug in a zkVM's constraint system lets a malicious prover forge proofs for executions that never happened. This is catastrophic in this setting, because the only artifact the attacker hands over is the proof itself, already accepted as valid by the verifier, so forgeries are invisible by construction. The risk is concrete: a 2018 soundness flaw in Zcash's BCTV14 system could have minted unbounded shielded ZEC against a then-$\sim$$1B market capitalization before the Sapling upgrade patched it [9], [10].

Internally, a zkVM pipelines program execution through trace generation, constraint encoding, proof generation, and verification (Section II-B). Bugs cluster around the constraint-encoding stage, both in the constraint system itself and in the executor that produces the trace it encodes; field arithmetic differs from 32-bit hardware in ways that force every operation to be reconstructed, and every reconstruction is a place where the encoding can silently diverge from the ISA (Section II-B). We call any such deviation a *semantic gap*, of two operationally distinct kinds: an *underconstraint* bug, where the constraints admit an alternative witness encoding a forbidden execution (a soundness violation exploitable by a malicious prover); and a *divergence* bug, where the executor itself drifts off-spec on a legitimate input (e.g., a crash on a valid program), yielding a completeness violation.

Detecting these gaps is hard for two reasons. First, the gaps cluster at non-trivial corner cases (division by zero, signed overflow, traces past 2^k table thresholds) that fuzzing without constraint-level guidance almost never reaches. Second, an underconstraint bug accepts an alternative trace that yields the same final registers, so output comparison alone misses it. Existing tools each address only part of the problem. Static analyzers (e.g., ZKAP [11]) target Circom-scale DSL circuits and miss zkVM-level instruction semantics; symbolic tools (e.g., Picus [12]) scale only to thousands of constraints, orders of magnitude below a production zkVM; prior dynamic testers (e.g., ARGUZZ [13]) combine metamorphic testing with random fault injection and target both kinds of gap, but without obligation-level guidance both the trace search and the injection step are blind. A generic RL-based fuzzer, the obvious modern baseline, fares no better: edge coverage in the Rust executor is uncorrelated with constraint structure, so its reward signal is uninformative; and blind witness-cell injection over a trace spanning millions of cells has negligible

Network and Distributed System Security (NDSS) Symposium 2027
22 - 26 March 2027, Seoul, Republic of Korea
ISBN xxx-x-xxxxxxx-x-x
https://dx.doi.org/10.14722/ndss.2027.xxxxxx
www.ndss-symposium.org

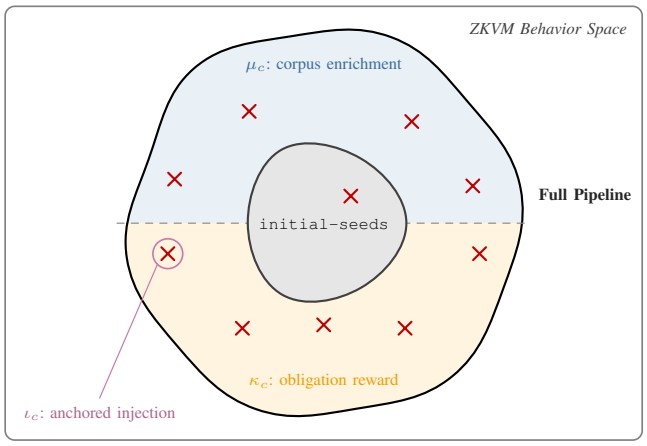

Fig. 1. Reachable behavior under a fixed exploration budget. The outer rectangle is the full *ZKVM behavior space*; the inner white region is what an unguided fuzzer reaches starting from `riscv-tests` [14] alone, finding only the bug (×) directly accessible from those seeds. BEAK's triple expands reach and triggers detection: μ_c (top) lifts the corpus onto obligation-active conditions; κ_c (bottom) replaces edge coverage with obligation-signature novelty; ι_c (purple rings) anchors witness injection at the activation step, surfacing underconstraint bugs invisible to output comparison. Bug positions are illustrative.

hit probability within any practical budget.

Our key insight is that these corner cases are not random: they correspond to *constraint obligations* derivable from two fixed sources: (i) the instruction-set specification, which dictates what each instruction must compute, and (ii) the algebraic structure of the proof system, which dictates how each computation must be encoded. Applied to RV32IM, this two-source derivation yields a closed set of 60 obligations (Section IV) shared across all leading production RV32IM zkVMs. We specify each obligation as a triple $(\mu_c, \kappa_c, \iota_c)$ of enrichment template, predicate, and injection template, derived once per obligation and reused across all backends.

The triple plugs into a generic RL fuzzing loop at the three places it would otherwise default to randomness, choosing seeds, scoring an execution, and probing the witness (Figure 1): μ_c constructs the initial corpus (replacing random seed selection), κ_c defines the bandit's reward via obligation-signature novelty (replacing edge coverage), and ι_c anchors witness probing at activation steps (replacing infeasible blind witness-cell injection). These three hooks pair with two detectors mapped 1-to-1 onto the bug taxonomy: differential register comparison against a RISC-V oracle catches divergences, while ι_c-anchored witness injection catches underconstraints invisible to output. Our ablation (Section VI-C) confirms each hook is necessary, with distinct roles: μ_c determines *which region* of the ZKVM behavior space the loop ever reaches (without it, the RL machinery has nothing obligation-relevant to amplify); κ_c *redirects exploration* from executor-state novelty to obligation-state novelty, recovering corner cases that standard edge coverage misses; and ι_c is *structurally required* for underconstraint bugs (the majority of our benchmark), which are invisible to any approach without obligation-anchored witness injection (a ceiling realized empirically by ARGUZZ's unguided fault injection at 5/37 on known bugs).

We implement this approach in BEAK, evaluated on all leading production RV32IM zkVMs (six systems spanning four distinct proof architectures). On a curated benchmark of 37 independently sourced known bugs, BEAK detects 26/37 (70.3%) versus ARGUZZ's 5/37 (13.5%), a 5.2× improvement, with BEAK's detections including all of ARGUZZ's. BEAK itself additionally discovers 14 previously unknown zero-day findings (all confirmed and patched). Most detections are underconstraint bugs that require directly probing the constraint system, which obligation-anchored witness injection makes tractable within practical budgets.

II. BACKGROUND

A. Zero-Knowledge Proofs

A *zero-knowledge proof* allows one party (the *prover*) to convince another party (the *verifier*) that a statement is true, without revealing any information beyond the truth of the statement itself. In the context of computation, the statement is: "I executed this program on this input and obtained this output." The prover produces a short cryptographic *proof* that the verifier can check in milliseconds, even if the original computation took hours. If the proof system is *sound*, no cheating prover can produce a valid proof for a false statement (e.g., a wrong output). Concretely, proof systems require the statement to be expressed as an *arithmetic circuit* (also called a *proof circuit*), a fixed network of addition and multiplication gates over a finite field whose evaluation encodes the computation being proved. Each application traditionally needed its own hand-crafted circuit, a labor-intensive and error-prone process that limited adoption to specialized use cases.

B. Zero-Knowledge Virtual Machines

A *zero-knowledge virtual machine* (zkVM) applies zero-knowledge proofs to general-purpose program execution. Rather than writing custom proof circuits for each application, a zkVM proves the correct execution of programs written in a standard instruction set, most commonly RISC-V, the open-source ISA used by all six systems we evaluate [2]–[7].

Figure 2 illustrates how a zkVM works, step by step:

1) **Execution.** The zkVM runs the RISC-V program instruction by instruction, recording every intermediate state (register values, memory contents, and internal control signals) in a table called the *execution trace*. Each row of the table corresponds to one instruction step.
2) **Constraint encoding.** The execution trace, together with any auxiliary intermediate values the prover commits to, forms a *witness* w. The witness is encoded as a system of polynomial equations A over a finite field $\mathbb{F}_p$ (the integers modulo a prime p); the constraint system *accepts* iff every equation in A evaluates to zero on w. Each equation asserts a correctness property of one or more trace rows. For example, for an `add` instruction at row i, the constraint system requires:

$$trace[i].rd = trace[i].rs1 + trace[i].rs2 \pmod{2^{32}}$$

This encoding is called the *arithmetization*. Crucially, $\mathbb{F}_p$ does not natively model 32-bit hardware: overflow wrap-around, sign extension, range, and bitwise operations all

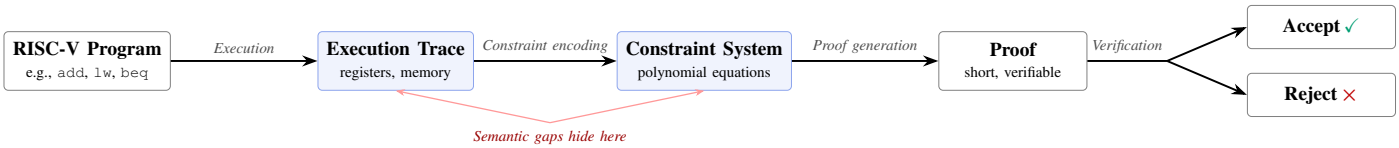

Fig. 2. How a RISC-V zkVM works. The prover takes a program and runs four stages (*Execution*, *Constraint encoding*, *Proof generation*, and *Verification*), each transforming one artifact into the next. Semantic gaps (deviations from the ISA in either the executor's trace or the constraint system) hide in the highlighted artifacts.

have to be *reconstructed* from field operations, and every reconstruction is a place where the constraint system can silently deviate from the ISA. All RISC-V zkVMs must enforce the same instruction semantics, regardless of how their internal trace tables are organized.

3) **Proof generation.** The prover uses a polynomial commitment scheme (e.g., FRI [15] or KZG [16]) to produce a succinct proof that all constraints are satisfied simultaneously.
4) **Verification.** The verifier checks the proof in polylogarithmic time, without access to the trace or the program's internal state.

C. Where Bugs Hide: Constraint Obligations

A production zkVM's constraint system contains hundreds of thousands of polynomial equations. We call any deviation between the constraint system or executor and the ISA a *semantic gap* (highlighted in Figure 2); gaps cluster at non-trivial correctness properties we call *constraint obligations*, rooted in two fixed sources: **(i)** the ISA specification (what the instruction must compute, e.g., `div` by zero, shift-amount masking) and **(ii)** the proof system's algebraic structure (how it is encoded over $\mathbb{F}_p$, e.g., 2^k-padded trace tables, limb decomposition for range checks). BEAK's methodology rests on this two-source derivation (Section IV). Two structural difficulties make these obligations hard to test:

- **Rare trigger conditions.** Obligations activate only under narrow conditions (division by zero, signed overflow, traces crossing 2^k thresholds) almost never exercised by standard test suites or unguided fuzzing.
- **The oracle gap.** A zkVM's own output cannot serve as ground truth, and the two failure modes call for different oracles. *Divergence* bugs (executor off-spec on a legitimate input, a completeness gap) yield observable wrong register state and admit differential testing against a trusted RISC-V emulator. *Underconstraint* bugs (executor correct but constraints too permissive, a soundness gap) leave the output identical and surface only as an alternative accepted witness; detecting them requires *witness injection*, directly mutating the prover's intermediate values and re-checking acceptance.

D. Reinforcement-Learning Fuzzing Baseline

Modern fuzzers cast mutation-operator selection as a multi-armed bandit problem: at each iteration the fuzzer picks among a pool of operators (bit flip, byte splice, dictionary insertion, etc.) and updates that operator's expected reward from observed outcomes. The UCB1 policy [17] is the de facto default: it balances exploration and exploitation with a closed-form upper-confidence bound and has provable logarithmic regret in stationary settings. BEAK uses a UCB1 variant with a tunable exploration constant and an ε-greedy override (Section V-D); we keep "UCB1" throughout when the distinction is immaterial. Beyond the policy itself, a generic RL fuzzer ships with three default choices that operate where domain knowledge is absent: a random or developer-supplied seed corpus, edge coverage in the executor binary as the reward signal, and (for tools that probe internal state) blind perturbation of intermediate values. All three defaults break on a zkVM (Section I); replacing each with a constraint-aware substitute is the central design problem BEAK addresses (Section III-A).

E. Threat Model

We adopt a *malicious prover* model: the prover knows the zkVM source code, witness generation logic, and constraint system, and aims to produce a proof accepted by an honest verifier for an execution trace that deviates from correct RISC-V semantics.

a) Soundness: Let O denote a reference RISC-V oracle and B a zkVM backend with executor E_B and constraint system A over $\mathbb{F}_p$. Backend B is *sound* if, for all efficient provers and inputs x, any accepted proof encodes a valid execution of x under O. A *soundness bug* is an input x with a witness w such that $A(w)$ accepts but w encodes a computation different from $O(x)$. We target bugs in the constraint system A and the executor E_B, not in the underlying cryptographic proof backend.

b) Completeness: Conversely, B is *complete* if every valid execution of x admits at least one witness w such that $A(w)$ accepts. A *completeness bug* is an input x for which the honest witness (the one E_B would produce in lockstep with O) is unavailable or rejected (e.g., $E_B(x)$ crashes or produces a register state inconsistent with $O(x)$). Both soundness and completeness bugs are semantic gaps and are in scope for BEAK.

c) Operational categorization: The two formal classes map 1-to-1 to two operational categories: soundness bugs are **underconstraint** (detected via witness injection at the obligation step), completeness bugs are **divergence** (detected via differential register comparison against O). This mapping underpins the dual-detector design (Section III-A) and the formal characterization in Section IV-C.

d) ISA scope: BEAK models RV32IM as specified in the RISC-V ISA manual [18]. Misaligned accesses trap; division follows the specification exactly (division by zero yields -1/dividend; signed overflow $-2^{31} \div -1$ yields -2^{31}/0).

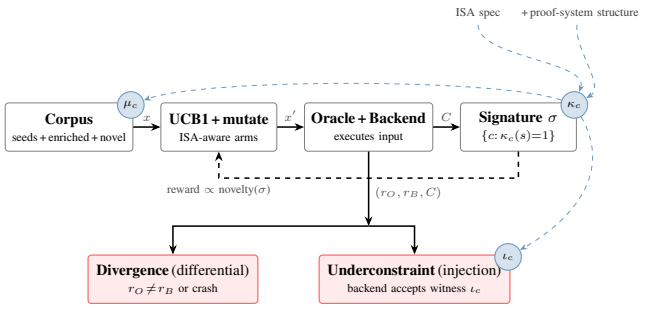

Fig. 3. Architecture of BEAK. **Derivation (top-right):** for each obligation c, the activation predicate κ_c is summarized from two sources (the ISA spec, which dictates what the semantics requires, and the proof-system structure, which dictates where activation can occur in the trace). The enrichment template μ_c (how to craft inputs that trigger κ_c) and the injection template ι_c (how to perturb the witness at κ_c-active steps) are then derived from κ_c. Dashed arcs visualize this chain. **Runtime loop:** the triple is not a separate stage; each component is a small badge *plugged into* the RL fuzzing loop at the place it acts: μ_c enriches the initial corpus (Section V-B); κ_c projects each post-execution trace into a signature σ whose novelty is the bandit reward (Section V-C); ι_c supplies the witness template used by the underconstraint detector at $(c, s) \in C$ (Section V-E). UCB1 over the ISA-aware arms (Table III) mutates $x \to x'$; Oracle and Backend execute and emit (r_O, r_B, C); both detectors consume this triple.

These corner cases are where several benchmark bugs manifest (Section VI-B).

III. OVERVIEW

We describe the framework architecture (Section III-A) and walk through a concrete bug as a running example (Section III-B); the formal characterization is in Section IV-C.

A. Framework Overview

Figure 3 illustrates BEAK's architecture: an RL fuzzing loop in which the obligation triple $(\mu_c, \kappa_c, \iota_c)$ replaces the three default-random loci of a generic fuzzer. Each of the 60 obligations enumerated by the semantic model (Section IV) supplies an enrichment template μ_c (inputs reaching the trigger condition), a predicate κ_c (recognizes activation on a trace), and an injection template ι_c (injects an alternative witness at the activation point).

a) The three obligation hooks: μ_c enriches each `riscv-tests` seed into an input that exercises one obligation from the first mutation iteration onward (Section V-B). κ_c projects each post-execution trace into an obligation signature whose novelty rewards the bandit arm that selected the mutation, replacing edge coverage (uncorrelated with constraint structure) with constraint-state novelty (Section V-C). ι_c anchors witness injection at the activation step (c, s) and probes whether the constraint system accepts an alternative encoding (Section V-E); without anchoring, the search is infeasible.

b) Bandit-guided ISA-aware mutation: At each iteration, a UCB1 multi-armed bandit selects an ISA-aware mutation arm (e.g., register swap, boundary-immediate substitution; full set in Table III) and applies it to a corpus entry (Section V-D). Mutation itself is obligation-agnostic: obligations enter the loop only after execution, through the κ_c-driven reward signal.

c) Oracle, backend, and two detectors: Each mutated input runs in parallel on a reference RV32IM simulator (yielding r_O; Section V-G) and on the instrumented backend (yielding r_B and the obligation activations C). Two detectors share the loop and target the two semantic-gap classes (Section II-E): differential register comparison flags *divergence* bugs ($r_O \neq r_B$ or executor crash; crashes routed through assertion-bypass instrumentation, Section IV-D); ι_c-anchored witness injection at $(c, s) \in C$ flags *underconstraint* bugs. The two detectors are orthogonal: divergence bugs leave no alternative witness for injection to find, and underconstraint bugs leave no register mismatch for differential to flag.

B. Motivating Example

We illustrate the three obligation-triple components with a real constraint bug in OpenVM's ALU chip (Figure 4), drawn from our 37-bug known benchmark of Section VI-B: a missing range check that lets a malicious prover claim an alternative encoding of a 12-bit immediate value that the ISA never permits.

a) Background: why immediates are split into bytes: Because $\mathbb{F}_p$ has no native notion of bit-width (Section II-B), every production zkVM splits each value into byte-sized chunks (*limbs*, a term from multiprecision arithmetic) that can be range-checked cheaply against a table of valid byte values; the 12-bit RISC-V immediate of `addi` is then *recomposed* from these limbs.

In OpenVM, the ALU chip splits each 12-bit immediate into two byte limbs $(limb_0, limb_1)$ and enforces a *recomposition constraint*:

$$limb_0 + limb_1 \cdot 256 = immediate \tag{1}$$

For instance, the immediate -1 (encoded as `0xFFF` in 12-bit two's complement) decomposes as $(limb_0, limb_1) = (\texttt{0xFF}, \texttt{0x0F})$, satisfying Equation 1. Because Equation 1 alone does not constrain how the prover chooses the limbs, the decomposition is unique only if the upper limb is range-checked to its actual 4-bit width via a range-check lookup:

$$0 \leq limb_0 < 256, \quad 0 \leq limb_1 < 16 \tag{2}$$

b) The bug: Consider the instruction `addi x1, x0, -1`, which sets register `x1` to -1 (i.e., `0xFFFF_FFFF` in unsigned 32-bit representation). OpenVM's ALU chip uses the recomposition (Equation 1) as the effective immediate without separately checking it against the decoded 12-bit value, and *omitted* the upper-limb range-check constraint (Equation 2) that bounds $limb_1 < 16$. With $limb_1$ unconstrained beyond byte width, a malicious prover can supply $(\texttt{0xFF}, \texttt{0x1F})$, which the constraint system accepts and recomposes to $\texttt{0xFF} + \texttt{0x1F} \cdot 256 = \texttt{0x1FFF}$. The ALU chip then uses `0x1FFF` in place of `0xFFF` when computing the instruction result, forging a proof for an execution that never happened.

c) How BEAK discovers this bug: The `alu.imm_limb` obligation specifies a triple: enrichment template μ ("replace immediates with boundary values"), predicate κ ("immediate crosses a limb boundary"), and injection template ι ("flip limb value"). Via μ_c, BEAK applies μ to a seed containing `addi`, producing `addi x1, x0, -1` as part of the initial corpus.

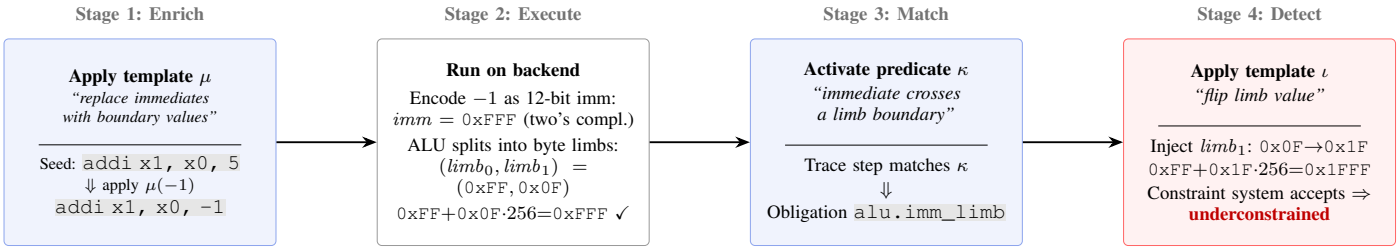

Fig. 4. BEAK discovers a missing range-check in OpenVM's ALU chip. The `alu.imm_limb` obligation specifies the triple (μ, κ, ι). Enrichment template μ produces a boundary-value immediate (-1) into the initial corpus; the backend trace activates predicate κ at the ALU step; the injection template ι flips the upper limb to an alternative decomposition (0x1FFF $\neq$ 0xFFF) that the constraint system accepts. The bug surfaces before any mutation iteration runs, because μ has already placed the trigger input in the corpus.

When the input runs, the backend trace activates predicate κ at the ALU step (matching κ_c). At that step, ι_c rewrites the limb pair to an alternative decomposition ($limb_1$: 0x0F $\rightarrow$ 0x1F) and re-evaluates the constraint system. The constraints accept the alternative, confirming an underconstrained witness. The bug surfaces before any mutation iteration runs, because μ has already placed the trigger input in the corpus; no combinatorial mutation is required.

d) Where the RL loop matters: Not all bugs yield to a single μ_c. When a bug requires the simultaneous activation of multiple obligations or a specific instruction-sequence pattern that no single enrichment template produces, κ_c and ι_c carry the load: as the bandit explores combinatorial mutations of the enriched corpus, κ_c rewards inputs that activate previously unseen obligation pairs, and once the right pattern fires, ι_c injects an alternative witness at the activation site and detects the underconstraint. Our ablation (Section VI-C) quantifies each component's contribution; the zero-day case studies in Section VI-D include underconstraints found through this combinatorial path rather than from the initial corpus directly.

IV. CONSTRAINT OBLIGATIONS

A correct RV32IM zkVM enforces the *same* correctness properties on the *same* RISC-V semantics. Register x0 must remain zero. Immediate values must decompose into correctly range-checked byte-sized pieces. Shift amounts must be masked to 5 bits. Memory timestamps must increase monotonically. We call each such property a *constraint obligation*: a correctness condition that every faithful zkVM must enforce. When a backend fails to enforce an obligation, a semantic gap exists: an underconstraint bug if the constraint system admits an alternative witness, or a divergence bug if the executor itself drifts off-spec on a legitimate input.

This insight is the foundation of BEAK's semantic model. Rather than analyzing each backend's constraint system directly, we systematically enumerate the constraint obligations that any correct RV32IM zkVM must satisfy. Each obligation c is defined by a *predicate* κ_c on a trace step that returns whether the obligation is activated; κ_c is the obligation's identity. The two other components of the triple, the *enrichment template* μ_c and the *injection template* ι_c, are mechanically derived from κ_c: μ_c inverts the predicate's trigger condition to construct inputs that satisfy it, and ι_c targets the witness cells named in the predicate to probe the constraint system. We refer to $(\mu_c, \kappa_c, \iota_c)$ as a *triple* because the three components plug into three distinct loci of the fuzzing loop (corpus enrichment, bandit reward, witness injection), even though only κ_c carries independent semantic content. Both derived components are consumed by the fuzzing loop in Section V; this section focuses on the obligations themselves, their derivation, predicate form, and coverage.

A. Obligation Derivation

We derive the 60 constraint obligations through a two-step top-down methodology: *functional decomposition* identifies the constraint components, and *obligation enumeration* identifies the correctness properties within each component.

a) Step 1: Functional decomposition: We organize the obligation set around the functional components commonly found in production RV32IM zkVMs: an instruction *decoder* (field extraction from 32-bit words), an *ALU* (arithmetic and logic), an *arithmetic unit* (multiplication and division), *control flow* (branches, jumps, system calls), an *execution pipeline* (operand routing and result binding), a *memory subsystem* (load/store consistency), *lookup tables* (range checks, boolean and XOR multiplicities), cross-component *interactions* (digest routing), *row management* (padding and table lifecycle), and a *timestamp model* (ordering guarantees). All ten components are present in each of the six target backends (Table I), giving 10 functional groups (Table II); a backend that omitted one of them would effectively leave an ISA function unconstrained, which is itself an underconstraint.

b) Step 2: Obligation enumeration: Within each functional group, we enumerate obligations along the two-source taxonomy of Section II-C:

- **Semantic** (ISA-rooted): x0 immutability (Decode), shift-amount masking to 5 bits (ALU), the division-by-zero result convention (Arithmetic), signed-vs-unsigned branch selection (Control), and similar properties whose violation makes the arithmetization accept a trace that encodes an incorrect RV32IM execution.
- **Algebraic** (STARK-rooted): every backend pads trace tables, segment tables, memory regions, lookup accumulators, commitment vectors, and range-check tables to powers of two (required by the FFT/NTT evaluation domain in STARK-based systems; analogous power-of-two structural constraints arise in Lasso's subtable decomposition and Nova's step-circuit padding), so crossing a 2^k threshold forces constraint-table reallocation that

exercises logic dormant at smaller sizes. These obligations live in the functional groups they affect (e.g., the memory table's 2^k capacity threshold is a Memory obligation, not a separate "resource" concern).

Both sources are enumerated uniformly. Table II lists all 10 groups, their counts, and one representative obligation with its predicate per group; the full enumeration of all 60 obligations and their linked benchmark bugs appears in Section A.

c) Non-triviality filter: At every scope of enumeration we apply a non-triviality filter: a candidate is admitted only when its underlying source (an ISA semantic edge case or a STARK-structural component) introduces non-trivial constraint complexity in the $\mathbb{F}_p$ embedding. Trivial cases are excluded by construction. The filter applies pervasively, at three scales:

- *Instruction level*: RV32IM features uniformly trivial across backends are not enumerated. For example, EBREAK and FENCE are implemented as halt/no-op across all six target backends with no constraint-system logic, and BEAK does not introduce obligations for them.
- *Operand level*: operand configurations that exercise no constraint complexity are not retained as test inputs. For example, `add x1, x2, 0` or mid-range non-boundary immediates exercise no per-limb decomposition stress and contribute no test value to any ALU obligation.
- *Trace-state level*: traces that do not approach algebraic boundaries are filtered before the obligations targeting those boundaries. For example, short programs that do not cross any 2^k threshold do not exercise table-padding obligations.

The filter is what makes systematic enumeration feasible: the raw configuration space is exponential, but its non-trivial subset is bounded by the structure of the ISA and STARK arithmetization combined.

d) Scope: Beyond the trivial cases removed by the filter above, two further classes are out of scope: implementation-specific properties such as backend-chosen memory range bounds, which are not ISA-level invariants and so are not enumerated; and the cryptographic proof backend (FRI / IPA / Lasso), which is assumed sound consistent with the threat model in Section II-E.

e) Running example: Consider `addi x1, x0, -1` executed on a production zkVM. The ALU component decomposes the 12-bit immediate (-1, encoded as `0xFFF`) into byte-sized pieces for constraint evaluation over a finite field. At this trace step, BEAK evaluates the `alu.imm_limb` predicate, which checks whether the immediate value crosses a byte boundary. The same predicate applies identically on any other backend processing the same instruction: obligations are defined on backend-agnostic trace-step properties, abstracting away how each backend internally represents the decomposition.

f) Formal definition: A trace step $s \in \mathit{Steps}(x)$ records the step index, instruction fields, and computed values (register state, immediate, memory address, etc.) during execution of input x on a backend. Each constraint obligation $c \in \mathcal{C}$ is identified by a predicate $\kappa_c : \mathit{Steps} \to \{0, 1\}$ that returns 1 when the trace step exercises the obligation. The *obligation signature* for input x is the set of all activated obligations:

$$\sigma(x) = \left\{ c \in \mathcal{C} \;\middle|\; \exists\, s \in \mathit{Steps}(x) : \kappa_c(s) = 1 \right\} \tag{3}$$

The signature is what the fuzzer observes about an input; Section V describes how it, together with each activated obligation's enrichment and injection templates, is consumed by the fuzzing loop.

Backend	Proving System	Architecture
SP1	STARK (Plonky3)	Modular chip AIR
Pico	STARK (Plonky3)	Monolithic STARK machine
Risc0	STARK (FRI)	Circuit-rv32im
Jolt	Lookup (Lasso)	Lookup-based verification
Nexus	Nova (IVC)	Incremental verification
OpenVM	STARK (Plonky3)	Modular chip AIR

Table I. Supported zkVM backends.

g) Predicate form: Each predicate κ_c is a pattern-matching rule on the trace step's instruction class and computed values, combining an instruction-class filter with a condition on the step's computed values (e.g., a value crossing a 2^k threshold, a register index at a special position, or a field element near the modulus). Table II gives one representative per group; the full set of 60 predicates follows this pattern (Section A).

h) Provenance and validation: The obligation set is derived top-down from the ISA specification and the STARK arithmetization, prior to and independently of any inspection of specific benchmark or zero-day bugs; no obligation was added or removed in response to a particular bug. We validate the set against 37 known bugs drawn from professional security audits, prior dynamic testing [13], and public issue trackers (with overlap; see Section VI-B). Of the 60 obligations, 31 are linked to at least one entry in the 37-bug known benchmark (Section A); the remainder come from specification analysis with no currently mapped known bug. The evaluation additionally includes 14 zero-day findings discovered by BEAK and held out from obligation derivation. The 11 missed known bugs require cross-component algebraic reasoning beyond single-step obligations, a limitation of the derivation granularity rather than of the enumeration within each component.

i) Coverage and extensibility: The 60 obligations cover the components present across our six target backends and, within each component, the correctness properties identified through ISA and arithmetization analysis: semantic obligations target ISA edge cases (bit-width truncation, special-case results, signed/unsigned disambiguation), and algebraic obligations target 2^n-threshold transitions at the tables a STARK-based backend pads. Each obligation is a single-step predicate, so bugs that depend on cross-component algebraic reasoning fall outside this framing; the 11 missed known bugs in our benchmark fall in this category. New obligations enter the framework as additional trace-step predicates κ_c, with no changes to the fuzzing loop, bandit, or witness injection.

B. How the Model Catches Bugs

Table II illustrates the model's operation. Beyond the motivating example (Section III-B), the algebraic obligations within

Group (#)	Example Obligation	Description	Predicate $\kappa_c(s)$
Decode (7)	`decode.zero_reg`	x0 immutability	$s.rd = \texttt{x0} \wedge s$ is a write
ALU (8)	`alu.shift_mask`	shift-amount masking	s is shift $\wedge\, s.shamt \geq 32$
Arithmetic (6)	`arith.div_rem`	division remainder bounds	s is div/rem $\wedge\, (s.divisor{=}0 \vee s.dividend{=}{-}2^{31})$
Control (8)	`control.branch_cond`	branch condition selection	s is branch $\wedge\, s.cmp\_signed \neq s.cmp\_unsigned$
Execution (7)	`exec.operand_bind`	source/dest operand binding	$s.rs1 = s.rs2$
Memory (12)	`mem.addr_align`	address alignment	s is memory $\wedge\, s.addr \bmod 4 \neq 0$
Lookup (5)	`lookup.boolean_mult`	boolean multiplicity	s is lookup interaction step
Interaction (2)	`interaction.digest`	cross-component digest routing	s is interaction $\wedge\, s.kind_1 \neq s.kind_2$
Row (2)	`row.padding_leak`	padding row interaction leakage	s is padding row
Time (3)	`time.boundary_origin`	timestamp boundary origin	s is first timestamp step (init row)

Table II. Constraint obligations by functional group, with one representative obligation and its predicate κ_c per group. Obligations within each group arise from ISA semantic requirements and STARK algebraic structure (2^k table thresholds); both sources are enumerated uniformly. The complete enumeration of all 60 obligations and their linked benchmark bugs is in Section A.

each group are particularly effective for divergence bugs: the Memory group's capacity threshold catches a memory-size crash at a 2^k address boundary, and the Execution group's segment boundary catches a RAM-size overflow at 2^k thresholds. In each case, the obligation names the condition and the detector confirms the bug within the same loop iteration. Further zero-day case studies are in Section VI-D.

C. Formal Properties of the Semantic Model

We formalize the properties that underpin the semantic model's detection guarantees and the complementary roles of the two detectors.

Definition 1 (Semantic Gap)**.** *Let B be a zkVM backend with executor E_B and constraint system A over field $\mathbb{F}_p$, let O be the reference RISC-V oracle, and let x be an input program. Write $r_t \in \mathbb{Z}_{2^{32}}^{32}$ for the register state after step t and $w_t \in \mathbb{F}_p^m$ for the witness row at step t (where m is the trace width). A* semantic gap *exists at instruction step t of x in B if at least one of the following holds:*

- ***Divergence:*** *The executor computes a register state that differs from the oracle: $E_B(x,t) \neq O(x,t)$, i.e., $\exists\, j \in [32] : r_{t,j}^B \neq r_{t,j}^O$. Equivalently, E_B deviates from the ISA on a legitimate x, so no spec-conforming witness exists for A to accept.*
- ***Underconstraint:*** *The executor computes the correct state ($E_B(x,t) = O(x,t)$), but the constraint system admits an alternative witness $w' \neq w_t$ at step t such that $A(w') =$ accept and $decode(w') \neq decode(w_t)$, where $decode : \mathbb{F}_p^m \to \mathbb{Z}_{2^{32}}^{32}$ extracts the register state encoded by a witness row. Equivalently, A over-approves and a malicious prover could substitute w' while still passing verification.*

Divergence gaps arise from prover-side (executor) errors; underconstraint gaps arise from verifier-side (constraint system) errors. The two classes are operationally disjoint by definition and map 1-to-1 to the two formal failure modes (completeness, soundness) of Section II-E. The *decode* function is backend-specific (e.g., one backend extracts 32 field elements and reduces modulo 2^{32}; another reconstructs from lookup table indices), but the definition is parametric in this mapping. The two classes require different detection mechanisms: divergence via output comparison, underconstraint via alternative-witness construction.

a) Dual-detector coverage characterization: Both phases run two detectors on every evaluation. Their coverage properties with respect to Definition 1 are as follows:

1) **The differential detector covers divergence gaps.** Register comparison detects all divergence gaps whose deviation propagates to the final register state, by construction (comparison is exact). Divergence gaps that crash the executor are handled via assertion-bypass instrumentation (Section IV-D), which converts fatal assertions to logged warnings and allows execution to continue to a (corrupted) final state.
2) **The injection detector covers underconstraint gaps within instrumentation reach.** Witness injection can detect underconstraint gaps at any step t for which an obligation activation exists at step s such that $-\delta_- \leq t - s \leq \delta_+$ (the asymmetric injection search window, default $(\delta_-, \delta_+) = (16, 64)$; the asymmetry reflects that algebraic effects of an alternative witness propagate forward through limb chains, lookup multiplicities, and timestamp ordering, leaving more candidate cells downstream of the activation than upstream). The coverage boundary is twofold: the window limits spatial reach, and the obligation model limits semantic reach (gaps at uninstrumented constraint components are invisible to injection).
3) **Neither detector subsumes the other.** Divergence gaps (executor errors) produce no alternative witness and thus cannot be detected by injection. Underconstraint gaps (correct executor, permissive constraints) produce no register mismatch and thus cannot be detected by differential comparison. This separation is structural: on the known benchmark, of the 26 bugs detected by BEAK, 9 are divergence-side and 17 are underconstraint-side, confirming that the two detectors are orthogonal capabilities targeting orthogonal classes.

b) Detector complementarity bound: On the 37 known bugs, removing the injection detector reduces BEAK's detections from 26/37 (70.3%) to at most 9/37 (the divergence bugs).

This bound is structural (it holds regardless of time budget, seed corpus, or mutation strategy) because underconstraint gaps produce no observable signal in differential comparison. This observation quantifies the precise contribution of witness injection and explains the performance gap between BEAK and prior dynamic testers such as ARGUZZ. ARGUZZ's fault injection targets the same underconstraint gaps; absent an obligation model to anchor the injection site, reaching equivalent coverage requires substantially more time and compute.

D. Backend Instrumentation

BEAK instruments each backend's trace-generation path via automated source patching in three passes: (1) **infrastructure injection** adds a uniform obligation-evaluation API; (2) **assertion bypass** replaces Rust `panic!`/`assert!` macros (identified by textual pattern matching on source code) with logged warnings, enabling execution of crash-inducing inputs (sandboxed execution with a watchdog mitigates the UB risk from bypassed assertions); (3) **predicate hooks** are inserted at each backend's constraint-component sites (ALU chips evaluate limb-decomposition predicates, memory chips evaluate address-alignment predicates, etc.).

The instrumentation is *sound* (obligation evaluations faithfully reflect computed trace facts) but not *complete* (new constraint components require new predicate hooks). Effort ranges from $\sim$200 to $\sim$600 LOC per backend; each backend exposes a uniform interface returning 32 registers and activated obligations. BEAK skips proof generation, reducing per-input evaluation from minutes to milliseconds. Table I lists the six supported backends.

V. OBLIGATION-TARGETED SEMANTIC FUZZING

Building on the semantic model of Section IV, this section details how the obligation triple plugs into the loop sketched in Section III-A: μ_c supplies the initial corpus (Section V-B), κ_c defines the bandit reward (Section V-C), and ι_c anchors witness injection (Section V-E). Crucially, obligation signatures are computed *after* execution, so they do not prescribe what to mutate: instead, novel signatures reward the bandit that selected the mutation arm, gradually steering future mutations toward underexplored constraint regions, while the same activations anchor injection.

A. The Fuzzing Loop

Algorithm 1 presents the loop as five procedures whose definitions are deferred to subsequent subsections. After INITIALCORPUS populates the corpus from an obligation-enriched seed set, each iteration mutates an input (MUTATE), executes it on the oracle and the instrumented backend (ORACLERUN, BACKENDRUN, returning r_O, r_B, and the activation list C), grows the corpus on novel obligation signatures, updates the bandit, and probes the witness at obligation-anchored steps (WITNESSPROBE). Two detectors share the loop on the same C: the **differential detector** (Section V-F) compares registers to flag divergences; the **injection detector** (Section V-E) runs WITNESSPROBE to flag underconstraints.

Algorithm 1 The BEAK Fuzzing Loop

procedure FUZZ(seeds S, backend B, oracle O, iterations N)
  $corpus \leftarrow \textsc{InitialCorpus}(S)$
  $Seen \leftarrow \emptyset$; $bugs \leftarrow \emptyset$; $\mathcal{B} \leftarrow \text{UCB1}(\{0, \ldots, 7\})$
  for $i = 1$ **to** N **do**
    $(x', a) \leftarrow \textsc{Mutate}(corpus, \mathcal{B})$
    $r_O \leftarrow \textsc{OracleRun}(x', O)$
    $(r_B, C) \leftarrow \textsc{BackendRun}(x', B)$
    if $r_O \neq r_B$ **then** ▷ **differential detector**: divergence
      $bugs \leftarrow bugs \cup \{(x', \texttt{"divergence"})\}$
    $\sigma \leftarrow \textsc{Signature}(C)$
    if $\sigma \notin Seen$ **then**
      $corpus \leftarrow corpus \cup \{x'\}$; $Seen \leftarrow Seen \cup \{\sigma\}$
    $\mathcal{B}.\textsc{update}\big(a, \textsc{Reward}(\sigma, Seen)\big)$
    $bugs \leftarrow bugs \cup \textsc{WitnessProbe}(C, B)$ ▷ **injection detector**: underconstraint
  return $bugs$

B. Initial Corpus via μ_c

$\textsc{InitialCorpus}(S)$ applies every μ_c (Section IV-A) to every seed in S (the `riscv-tests` suite, Section V-H), placing inputs that already exercise each of the 60 obligations into the corpus before iteration 1. `riscv-tests` programs do not exercise immediates at byte boundaries, traces past 2^k thresholds, or similar obligation-critical conditions; random mutations rarely produce them either. μ_c shifts these conditions from "must be discovered" to "available before iteration 1," letting the bandit spend its budget on combinatorial mutations of obligation-active inputs.

C. Bandit Reward via κ_c

BEAK replaces edge coverage (e.g., AFL [19]), which targets executor branches uncorrelated with constraint structure, with *signature novelty* computed from $\{\kappa_c\}_{c \in \mathcal{C}}$: an input is added to the corpus iff its signature $\sigma(x)$ (Equation 3) is unseen. κ_c projects each trace into a 60-dimensional constraint-coverage vector that captures exactly the structure where bugs hide. The bandit reward in Algorithm 1 is fine-grained:

$$\textsc{Reward}(\sigma, Seen) \;=\; \mathbb{1}\big[\sigma \notin Seen\big] \;+\; 0.25 \cdot \big|\sigma \setminus SeenObls\big|,$$

where $SeenObls = \bigcup_{\sigma' \in Seen} \sigma'$ aggregates the individual obligation IDs activated by any signature observed so far. The first term rewards novel signature *combinations* (weight 1.0); the second provides a gentler signal for inputs that activate previously unseen *individual* obligations (weight 0.25 each).

D. ISA-Aware Bandit-Guided Mutation

$\textsc{Mutate}(corpus, \mathcal{B})$ chooses an input from the corpus, selects a mutation arm via the bandit $\mathcal{B}$, applies the corresponding ISA-aware mutation, and returns the new input together with the chosen arm.

a) Mutation strategies: BEAK employs 8 mutation strategies, each operating at the RISC-V instruction level rather than the byte level (Table III). Unlike generic byte-level

Arm	Strategy	Description
0	Splice	Cross two corpus entries at a random point
1	Register swap	Replace rd/rs1/rs2 with a register already used in the program
2	Constant mutation	Replace immediates with boundary values
3	Insert instruction	Append a random instruction; prefer memory ops when address-producing registers exist
4	Delete instruction	Remove one instruction from the sequence
5	Duplicate instruction	Copy an instruction to increase data locality
6	Swap adjacent	Exchange two neighboring instructions
7	Mnemonic replace	Substitute with a same-format mnemonic (e.g., `add`↔`sub`, `lw`↔`lh`)

Table III. ISA-aware mutation strategies. Each arm operates on decoded RISC-V instruction fields, maintaining encoding validity while targeting specific semantic dimensions. Arms 1, 2, 3, and 7 embed obligation awareness.

mutations (e.g., bit flips, byte swaps), these strategies understand the RV32IM encoding format: they decode instructions into fields (opcode, rd, rs1, rs2, funct3, funct7, immediate), mutate semantically meaningful components, and re-encode valid instructions. This ISA awareness is critical because random byte-level mutations on 32-bit instruction words almost never produce valid RISC-V instructions, let alone programs that exercise interesting constraint behaviors. The ISA-aware encoder/decoder is validated against the official `riscv-tests` suite (all RV32IM encodings round-trip correctly).

b) Obligation-arm alignment: Obligations influence mutation through two complementary mechanisms: *static design* embeds domain knowledge into specific arms, while *dynamic adaptation* lets the bandit learn which arms are productive per backend.

Four arms encode direct obligation awareness. Arm 1 (register swap) can redirect `rd` to `x0`, directly triggering the `decode.zero_reg` obligation (the mechanism behind the x0 zero-days discovered across three backends), and can set `rs1=rs2`, triggering `exec.operand_bind`. Arm 2 (constant mutation) replaces immediates with boundary values that stress both byte-piece decomposition (ALU, Arithmetic obligations) and 2^k-threshold transitions (Memory, Execution, Lookup obligations). Arm 3 (insert instruction) biases toward memory operations when address-producing registers exist, driving memory footprint toward 2^k capacity thresholds (Memory, Row obligations). Arm 7 (mnemonic replace) tests opcode-selector constraints by substituting same-format instructions (e.g., `add`↔`sub` for ALU obligations, `beq`↔`blt` for `control.branch_cond`, `lw`↔`lh` for Memory sign-extension obligations).

The remaining four arms (0: splice, 4: delete, 5: duplicate, 6: swap adjacent) are structurally generic: they create input diversity without targeting any specific obligation. Their value is that they produce novel instruction sequences that the obligation-aware arms alone would not reach.

Obligations do *not* prescribe mutation at runtime; the obligation signature is computed *after* execution. The bandit's reward signal (Section V-C) closes the feedback loop instead: it teaches the bandit which arms are most productive for each backend's constraint structure, so that obligation-aware arms are upweighted when they consistently trigger novel activations, and generic arms are retained when they contribute structural diversity that the targeted arms miss.

c) UCB1 selection: BEAK uses a UCB1 variant [17] with a tunable exploration constant and an ε-greedy override. At each iteration, with probability $1-\varepsilon$ the arm with the highest upper confidence bound is selected,

$$a^* = \arg\max_{a \in \{0,\ldots,7\}} \left(\bar{r}_a + \alpha \sqrt{\frac{\ln n}{n_a}} \right),$$

where $\bar{r}_a$ is arm a's mean reward, n_a is its pull count, n is the total pull count, and $\alpha = 1.5$ is the exploration constant; with probability $\varepsilon = 0.05$, a uniformly random arm is selected instead, ensuring all arms continue to be exercised after UCB1's confidence bounds have converged. The α-tuning and ε-override depart from textbook UCB1 (which fixes $\alpha = \sqrt{2}$ and uses no ε); both adjustments were retained because they consistently improved time-to-finding on a held-out tuning set disjoint from the 37-bug evaluation benchmark.

E. Witness Probe via ι_c

Underconstraint bugs do not manifest as register mismatches: they allow the prover to substitute an *alternative witness* that the circuit still accepts. WITNESSPROBE(C, B) detects them using each activated obligation's injection template ι_c, which makes the search feasible by anchoring it at the obligation-active step rather than across the whole trace.

a) Procedure: *injection_targets*(C) maps activated obligations to (c, s) anchor pairs. For each pair, ι_c generates candidate modifications w' at trace steps within a window $[s - 16,\ s + 64]$ around the anchor, capped at $T_{\max} = 64$ trials per obligation. For every candidate with $w' \neq w_s$, BEAK invokes the backend's CONSTRAINT_CHECK(w') function and flags the input as an *underconstrained candidate* if the constraint system accepts. WITNESSPROBE returns the set of such flagged inputs.

b) Direct algebraic check: Each backend exposes a CONSTRAINT_CHECK function that evaluates its AIR/RAP constraints on the (possibly modified) trace and returns whether all constraint polynomials evaluate to zero. The check covers both *boundary constraints* (single-row conditions such as range checks and boolean constraints) and *transition constraints* (adjacent-row conditions such as register-file consistency and memory timestamp ordering) by supplying the modified row along with its neighbors. This is a *direct algebraic check* on the constraint system: it does not require generating a full proof (which would take minutes), only evaluating the local constraint window (which takes microseconds). Cross-table constraints enforced via lookup or permutation arguments (e.g., Lasso or LogUp in various backends) are *not* checked, since verifying these requires a global accumulator pass over the entire trace; the rare false positives this admits are filtered by post-injection output comparison and we flag this as a known limitation. The injection type is determined by the obligation's functional group: flipping a byte-piece value for

ALU obligations, altering an address for memory obligations, toggling a boolean for lookup obligations, or perturbing a 2^k boundary value for obligations at power-of-two table thresholds (e.g., memory capacity, segment boundary, range-table size).

c) Soundness classification and limitations: An injection is a true positive if (1) the modification is non-trivial ($w' \neq w$), (2) the constraint system accepts w', and (3) w' encodes a computation different from RISC-V semantics. Condition (3) is satisfied by design: each injection type produces a semantically different result (flipped byte-pieces change integer values, toggled booleans change control flow). Rare semantically-equivalent alternatives are filtered by post-injection output comparison.

The patterns are heuristic single-field perturbations (byte-piece flips, address alterations, boolean toggles, 2^n boundary shifts) at individual constraint steps. The 11 missed bugs require *multi-field, cross-component* modifications beyond current patterns. Injection tests only local effects; propagated effects are caught only if downstream divergence manifests in the differential detector or triggers a separate injection.

F. Differential Detector

The differential detector compares the 32 architectural registers returned by ORACLERUN and BACKENDRUN: any input with $r_O \neq r_B$ is flagged as a divergence bug. Register comparison detects every divergence gap whose deviation propagates to the final register state, by construction. Crashes are routed through the assertion-bypass instrumentation (Section IV-D), which converts fatal Rust assertions to logged warnings so the executor continues to a (corrupted) final state that remains comparable to the oracle's. We compare registers rather than full memory to avoid normalizing heterogeneous address-space layouts; memory-only divergences are partially addressed by witness injection at memory-related constraint steps.

G. Reference Oracle

ORACLERUN(x', O) executes x' on a reference RV32IM simulator built on `rrs-lib`, validated against the official `riscv-tests` compliance suite, and returns the post-execution architectural register state r_O. It supports two memory models (`SharedCodeData` and `SplitCodeData`), selected per backend to match the backend's architecture and avoid false positives. Edge cases follow the RISC-V specification: `ECALL` is a no-op recording arguments, `EBREAK` traps, `FENCE` is a no-op, and misaligned accesses trap (flagged as a divergence bug if the backend handles them silently). The oracle also pre-filters inputs exceeding a configurable step limit.

BACKENDRUN(x', B) executes x' on the instrumented backend (Section IV-D), returning the same register state r_B together with the list C of obligation activations evaluated during trace generation.

H. Implementation

BEAK is approximately 12,000 lines of Rust (backend instrumentation per Section IV-D, obligation evaluation, witness injection, ISA-aware mutation) and 3,000 lines of Python (RL loop, bandit scheduling, corpus management). Per-backend instrumentation effort ranges from $\sim$200 to $\sim$600 LOC. The base seed set is 2,172 `riscv-tests` sequences (32-bit instruction words, up to 27 instructions per seed; Phase 2 mutations capped at 256 instructions); the seeds cover all RV32IM operations including edge cases (division by zero, signed overflow, misaligned memory). Proof generation is skipped: each iteration evaluates only trace generation plus the local constraint-check (AIR/RAP polynomials on the modified row and its neighbors), reducing per-input cost from minutes to milliseconds.

VI. EVALUATION

We evaluate BEAK to answer the following research questions:

- **RQ1 (Effectiveness):** How effective is BEAK at discovering semantic gaps (divergence and underconstraint) compared to state-of-the-art zkVM testing approaches?
- **RQ2 (Ablation):** What is the contribution of each obligation hook: seed enrichment, obligation-signature reward, and witness injection?
- **RQ3 (Zero-Days):** Can BEAK discover previously unknown vulnerabilities in production-grade zkVMs?

A. Experimental Setup

a) Target zkVMs: We evaluate BEAK on six production-grade RISC-V zkVMs spanning four proof architectures (STARK/Plonky3, STARK/FRI, Lookup/Lasso, Nova/IVC): SP1, Pico, OpenVM, Risc0, Jolt, and Nexus (Table I). For each system, we test against one or more pinned commit snapshots to ensure reproducibility.

b) Seed corpus and obligation enrichment: The base seed set is 2,172 `riscv-tests` sequences (Section V-H); μ_c-enrichment then constructs the loop's initial corpus (Section V-B).

c) Baselines (for RQ1): RQ1 compares BEAK against two external baselines; the ablation variants of BEAK itself are introduced separately in RQ2 (Section VI-C).

- **ARGUZZ** [13]: a state-of-the-art dynamic testing framework for zkVMs that combines metamorphic testing with fault injection. We re-ran ARGUZZ from its public artifact on the same hardware, with the same per-backend time and core budget as BEAK, against the same six backends and the same 37-bug benchmark.
- **Baseline**: a generic fuzzer with uniform random byte-level mutation and differential register comparison only; it lacks semantic feedback, obligation guidance, ISA-aware operators, and witness injection. It serves as the unguided floor establishing that the benchmark is non-trivial.

d) Configuration: All experiments are conducted on an AMD EPYC 9654 server with 96 cores / 192 hardware threads, 768 GB RAM, and 2×4 TB NVMe storage, running Ubuntu 22.04. Per-backend campaigns are allocated 32 CPU cores each and run sequentially with a 128-hour timeout per campaign; the same allocation is used for all baselines. To control for stochastic variance from the bandit and randomized mutations, we run each (tool, backend) campaign multiple times and

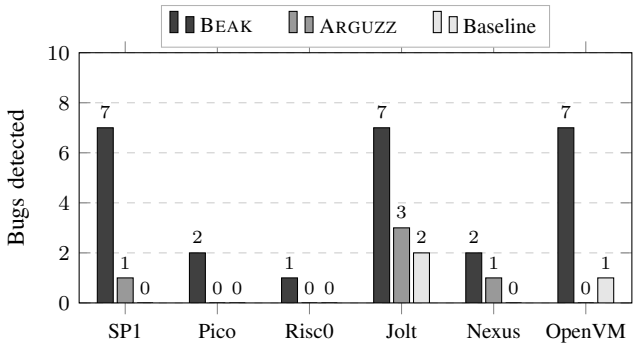

Fig. 5. Bug detection per backend on the 37-bug known benchmark (per-backend totals: SP1 9, Pico 2, Risc0 5, Jolt 7, Nexus 3, OpenVM 11). Across all backends, BEAK detects 26/37, ARGUZZ 5/37, and Baseline 3/37; BEAK's detections include all of ARGUZZ's.

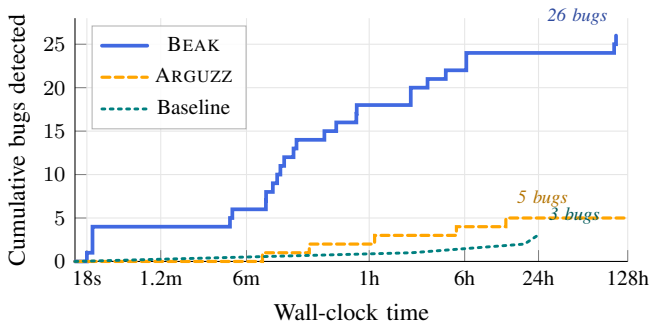

Fig. 6. Cumulative unique known bugs over time. BEAK reaches a higher detection rate than ARGUZZ and Baseline under the same finite-time reproduction criterion.

report the union of distinct bugs detected across runs, following recommended practice for stochastic fuzzing evaluation [20]. BEAK uses the default configuration: UCB1 with exploration constant $\alpha = 1.5$ and an ε-greedy override at $\varepsilon = 0.05$, witness injection asymmetric search window of 16 steps before and 64 steps after the anchor, and a maximum of 64 injection trials per obligation.

e) Metrics: We measure two metrics: (1) **unique bugs found**, the count of distinct divergence and underconstraint candidates; and (2) **unique obligation signatures**, the number of distinct signature combinations discovered.

B. RQ1: Bug Detection Effectiveness

To assess BEAK's effectiveness, we construct a ground-truth benchmark of 37 independently sourced known zkVM semantic-gap bugs, separate from the 14 zero-day findings discovered by BEAK. The known benchmark is drawn from three sources: 11 bugs previously reported by ARGUZZ [13], 25 bugs from professional security audits of production zkVMs [21]–[29], and 8 bugs from public issue trackers (with overlap: some bugs appear in multiple sources). The benchmark partitions cleanly along the divergence/underconstraint axis introduced in Section IV-C: 28 are underconstraints (the executor computes the correct state but the constraint system admits an alternative witness), and 9 are divergences (the executor itself drifts off-spec on a legitimate input). For each bug, we determine whether each tool can detect it through the corresponding detector (register-level comparison for divergence or witness injection for underconstraint). Figure 5 presents per-backend detections; time-to-finding dynamics are reported separately in Figure 6.

On the 37 known bugs, BEAK detects 26/37 (70.3%); ARGUZZ detects 5/37 (13.5%) and Baseline detects 3/37 (8.1%). BEAK detects all five finite-time ARGUZZ hits and all three Baseline hits, despite using a fundamentally different oracle (differential register comparison plus witness injection vs. metamorphic relations or unguided mutation). The 26 detections split into 17 underconstraint and 9 divergence bugs: all 9/9 divergence bugs are detected (the differential detector saturates at the benchmark's divergence ceiling), and 17/28 underconstraint bugs are detected via ι_c-anchored witness injection. The majority of BEAK's signal beyond the divergence ceiling thus comes from witness injection.

a) Per-backend analysis: Detection counts vary with constraint architecture: Jolt (7/7) and Pico (2/2) cover all known benchmark bugs in their backends, while SP1 (7/9), Nexus (2/3), and OpenVM (7/11) mix localized errors with deeper cross-component issues. Risc0 (1/5) remains limited: its circuit-rv32im architecture encodes witness columns in a non-canonical interleaved layout, making ι_c anchoring imprecise; four of the five Risc0 benchmark bugs require multi-column modifications that cross circuit boundaries. The 11 missed known bugs share a structural pattern: they require *cross-component algebraic reasoning* beyond single-instruction constraint obligations or currently unmodeled witness structure.

Result for RQ1: On the 37 known bugs, BEAK detects 26/37 (70.3%) versus ARGUZZ's 5/37 (13.5%), a 5.2× improvement; BEAK's detections include all of ARGUZZ's. Baseline detects 3/37 (8.1%).

b) Temporal dynamics: Figure 6 shows cumulative bug discovery over wall-clock time on the 37-bug known benchmark. BEAK's curve rises steeply, reaching 24 of 26 detected known bugs within 6.2 hours, and then finds two long-tail Jolt bugs at roughly 99.2 and 102.8 hours. Both long-tail bugs require specific multi-instruction patterns (2^k-aligned trace lengths combined with rare lookup interactions) that the bandit eventually rediscovers through stochastic exploration; under a 96-hour budget the result would be 24/37 rather than 26/37. ARGUZZ reaches 5 detected known bugs by 13 hours under the same finite-time reproduction criterion. Baseline reaches 3 detected known bugs by roughly 23.6 hours and does not cover any verifier-side witness-injection case beyond normal-execution cases reachable by byte-level mutation.

C. RQ2: Ablation Study

We ablate BEAK's obligation hooks by replacing each with the default-random mechanism it displaces. All ablation variants below retain BEAK's ISA-aware mutation operators (Table III); they differ only in which obligation hook is removed.

- **BEAK** *: removes specification-driven enrichment μ_c and starts from raw `riscv-tests`.

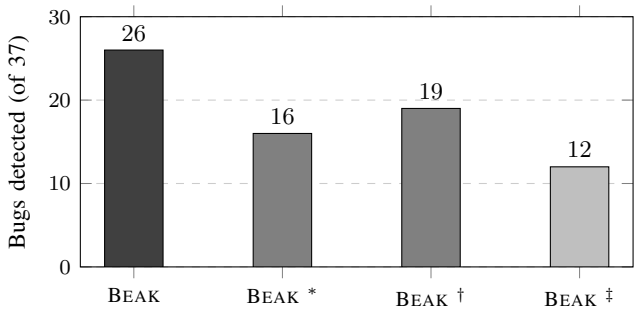

Fig. 7. Ablation study on the 37 known bugs under the 128-hour-per-backend budget. BEAK is the full pipeline. * removes specification-driven enrichment μ_c; † replaces obligation-signature reward κ_c with edge coverage; ‡ replaces all three obligation hooks with their default-random alternatives. External baselines (ARGUZZ, Baseline) are reported in RQ1 (Figure 5).

- **BEAK** †: replaces obligation-signature novelty κ_c with edge-coverage reward.
- **BEAK** ‡: removes all three hooks (raw seeds, edge-coverage reward, blind unanchored injection); the all-hooks-off reference point.

Figure 7 reports detections on the 37 known bugs under the same 128-hour-per-backend budget as RQ1.

a) Effect of obligation guidance: The ablations show that the three hooks contribute complementary signal. Removing μ_c drops detections from 26/37 to 16/37: the raw seed corpus exercises legal ISA behavior, but only sparsely reaches obligation-critical conditions such as byte-boundary immediates or 2^n table thresholds. Replacing κ_c with edge coverage yields 19/37, indicating that executor-state novelty is an imprecise proxy for constraint-state novelty.

b) Effect of anchored injection: The injection hook is not meaningfully isolated by a single-hook blind variant: production traces contain millions of witness cells, whereas a bug-relevant obligation neighborhood spans only a small local window. This search asymmetry makes blind witness-cell injection infeasible at the evaluated budget. Consistent with this structural argument, BEAK ‡, which removes all obligation guidance and uses blind unanchored injection, detects only 12/37, and the full pipeline more than doubles this no-obligation variant, reaching 26/37. An injection-only variant retaining ι_c but replacing μ_c and κ_c with defaults would in principle isolate injection's contribution; without κ_c to identify active obligations, ι_c has no anchor points and degrades to blind injection.

Result for RQ2: Each obligation hook is load-bearing. Removing seed enrichment, obligation-signature reward, or all hooks reduces detection to 16/37, 19/37, and 12/37, respectively; the full pipeline reaches 26/37.

D. RQ3: Zero-Day Discovery

We applied BEAK to the latest development branches of six zkVM systems and discovered 14 zero-day findings. These include critical soundness violations enabling proof forgery, CPU/GPU trace-divergence bugs, and production-crashing precompile failures. All were responsibly disclosed, confirmed, and patched.

a) Case study: x0 register overwrite: The same vulnerability class exists independently in three backends: one allowed an ecall to write to x0; another lacked a circuit-layer zero-register constraint; a third aliased registers with memory, so `sw` to address 0 overwrites x0. All three map to the `decode.zero_reg` obligation. BEAK automatically detects the memory-aliasing instance through its standard fuzzing loop; the other two were identified through manual inspection guided by the obligation model and confirmed by the respective maintainers.

b) Case study: `is_real` *bypass and timestamp wrap:* In one backend, the `MemoryLocalChip` omits the boolean constraint on the `is_real` lookup multiplicity; the `lookup.boolean_mult` obligation anchors injection of a non-boolean value, and the circuit accepts. In the same backend, the timestamp monotonicity check wraps modulo $p \approx 2^{31}$; the `time.boundary_origin` obligation anchors injection of a near-p timestamp. Both are underconstraints, invisible to differential testing.

c) Security impact: The critical underconstraint zero-days enable forging proofs for incorrect RISC-V execution. The x0 vulnerability is particularly noteworthy: the same ISA-level invariant was independently violated in three unrelated codebases.

Result for RQ3: 14 zero-day findings across the six backends, all confirmed and patched. Three share a common semantic class (x0 overwrite), demonstrating cross-backend generality of the semantic model.

VII. RELATED WORK

a) Zero-knowledge proof security: ZKAP [11] applies static analysis to Circom circuits; Picus [12] verifies Circom witness uniqueness via SMT; zkFuzz [30] (IEEE S&P 2026) uses mutation-based dynamic analysis for Circom circuits, combining trace characterization with joint program/input mutation and an error-based fitness signal; Coda [31] verifies ZK circuit implementations against high-level specifications via refinement types and Coq-backed proof obligations. All four target DSL-level circuits or circuit programs rather than instruction-set-level zkVM semantics. CircUZZ [32] (CCS 2025) and MTZK [33] (NDSS 2025) apply metamorphic testing to ZK *compiler pipelines*: CircUZZ tests four pipelines (Circom, Corset, Gnark, Noir) by stacking equivalence-preserving circuit rewrites, while MTZK designs metamorphic relations targeting ZK compiler correctness. Both operate at the DSL-to-circuit compilation boundary and detect compiler-introduced bugs; neither addresses zkVM-level constraint soundness or RISC-V semantic fidelity. zkVM arithmetizations differ in two ways: *scale* (10^5–10^6 AIR/lookup constraints with cross-linked trace tables over small-characteristic fields such as BabyBear, KoalaBear, and Goldilocks, beyond SMT solver capacity in practice) and *specification* (verifying soundness against underconstraint requires modeling both the constraint system and RISC-V semantics; we are unaware of a production zkVM that exposes such a specification). FORAY [34] synthesizes DeFi attacks at the application level; Tabby [35] optimizes circuit *efficiency* via program synthesis, complementary to BEAK's focus on *correctness*.

ARGUZZ [13] is the closest work: it applies metamorphic testing with fault injection to zkVMs and targets both prover-side divergence and verifier-side underconstraint gaps. ARGUZZ searches the trace without an obligation model, so its fault injection is untargeted; reaching underconstraint sites that BEAK covers via ι_c-anchored injection requires substantially more time and compute. Concretely, on the motivating example of Section III-B, ARGUZZ is unlikely to generate immediate -1 *and* exercise the specific ALU limb-decomposition path within a practical budget, while ZKAP [11], operating on Circom-level static analysis, lacks the trace-table structure needed to reason about limb-recomposition consistency at all. BEAK's design adds two structural pieces: (1) the obligation triple $(\mu_c, \kappa_c, \iota_c)$ restructures the three default-random decision points of the fuzzing loop (initial corpus, RL reward, witness probe) so that constraint corner cases are exercised systematically, and (2) witness injection anchored at κ_c-active steps via ι_c reduces the per-obligation probe to a tractable per-step check. On the known benchmark, BEAK detects 26/37 to ARGUZZ's 5/37 ($5.2\times$) under the same finite-time reproduction criterion, and BEAK's detections include all of ARGUZZ's. The additional bugs are predominantly underconstraints, consistent with the structural advantage of obligation-anchored injection.

b) Domain-specific coverage metrics: BEAK belongs to a growing trend replacing edge coverage with domain-specific metrics. Data Coverage [36] (USENIX Security 2024) tracks constant-data references as fuzzing feedback, exposing behavior that code coverage alone can miss. zkFuzz extends this idea inside individual Circom circuits via trace characterization and error-guided mutation; BEAK extends it from DSL-level circuits to RV32IM zkVMs along two axes: (1) obligations form a triple that simultaneously seeds the corpus (μ_c), defines the reward (κ_c), and anchors witness injection (ι_c), so a single specification artifact drives three loop decisions instead of feeding a single coverage metric; and (2) the reward combines novel obligation-signature *combinations* with a per-obligation bonus for newly activated obligations (Section V-C), giving more sustained signal than flat set-based coverage in the large but repetitive constraint spaces of zkVMs.

c) Differential testing, fuzzing, and mutation scheduling: Differential testing [37], [38] compares outputs of multiple implementations; BEAK adapts this to zkVMs with obligation-signature novelty feedback. Coverage-guided fuzzers (AFL [19], LibFuzzer [39]) use edge coverage, inadequate for zkVMs where interesting behaviors are at the constraint level; Klees et al. [20] further show that sound empirical comparison requires controlled statistical methodology, which we adopt. Grammar-aware fuzzers (Nautilus [40], Gramatron [41]) ensure structural validity; BEAK's ISA-aware mutations additionally incorporate constraint-obligation awareness. MOpt [42] and EcoFuzz [43] apply optimization-based scheduling to mutation operators; NEZHA [44] uses δ-diversity for differential testing; SemFuzz [45] extracts semantic information from CVE descriptions. BEAK is unique in this space in that the same obligation triple simultaneously seeds the corpus (μ_c), defines the bandit reward via signature novelty (κ_c), and anchors witness injection (ι_c); the three default-random decision points of a generic RL fuzzer are all replaced by obligation-driven structure.

VIII. CONCLUSION

We presented BEAK, an RL fuzzing framework for finding semantic gaps in RISC-V zkVMs spanning both soundness violations (underconstraint) and completeness violations (divergence). BEAK models constraint obligations as triples $(\mu_c, \kappa_c, \iota_c)$ of enrichment template, predicate, and injection template, and defines 60 such obligations across 10 functional components. The triple's three components hook into the three default-random decision points of an RL fuzzing loop: μ_c supplies the initial corpus, κ_c defines the bandit reward via signature novelty, and ι_c anchors witness injection at activation steps. Without these substitutions, RL fuzzing on a zkVM is structurally infeasible (the seed corpus exercises only common-path constraints, edge coverage in the executor is uncorrelated with constraint structure, and blind witness-cell injection cannot find soundness gaps in traces spanning millions of cells). On a curated benchmark of 37 known bugs across all leading production backends, BEAK detects 26/37 (70.3%) versus ARGUZZ's 5/37 (13.5%), a $5.2\times$ improvement; detections include all of ARGUZZ's. BEAK additionally discovered 14 zero-day findings during the same evaluation, all responsibly disclosed and patched.

Three findings emerge from the evaluation. First, verifier-side underconstraints dominate over prover-side divergences at a 28:9 ratio in the known benchmark; the majority of detections are invisible to differential-only approaches, validating obligation-anchored witness injection as a first-class detection mechanism. Second, the obligation model generalizes: the benchmark bugs map to obligations derived from specification analysis rather than post hoc pattern matching, and the same triple abstraction applies uniformly across all leading production backends. Third, the three obligation hooks are individually load-bearing: ablating κ_c as the reward signal drops detections from 26 to 19 bugs; the ι_c anchoring is structurally inseparable from injection itself (blind witness-cell injection over traces spanning millions of cells is computationally infeasible). The undetected bugs require cross-component algebraic reasoning beyond single-step obligations; extending the model with interaction-level obligations is the most promising direction. We will open-source BEAK upon acceptance.

ETHICS CONSIDERATIONS

This work uncovers soundness vulnerabilities in production zero-knowledge virtual machines (zkVMs) that secure billions of dollars in on-chain value. We took the following measures to ensure responsible conduct.

a) Responsible disclosure: All 14 zero-day findings discovered by BEAK were privately disclosed to the maintainers of the affected zkVMs through their established security channels (security email addresses or private bug-bounty platforms) before any public communication. We coordinated with each maintainer on triage, root-cause analysis, and remediation. Every finding has been confirmed by the maintainers and patched in the latest released versions of the affected systems prior to submission of this paper. We do not release any proof-of-concept exploit that could be weaponized against deployed systems.

b) Local-only experimentation: All experiments were conducted on local instances of the open-source zkVM implementations using synthetic inputs. We did not interact with any production deployment, mainnet rollup, or live blockchain bridge. No real user funds, transactions, or private data were involved at any point.

c) Research artifacts: The benchmark of known bugs we use is drawn from publicly disclosed audit reports and prior academic work; we do not redistribute non-public information. Our framework, obligation specifications, and reproduction scripts will be released after the patches we triggered have had time to propagate to downstream consumers, in line with standard practice for offensive-security tooling whose disclosure could otherwise enable attacks against unpatched forks.

REFERENCES

[1] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, ser. STOC '85. New York, NY, USA: Association for Computing Machinery, 1985, p. 291–304. [Online]. Available: https://doi.org/10.1145/22145.22178

[2] RISC Zero, "RISC Zero: General-purpose verifiable computing," https://github.com/risc0/risc0, 2024.

[3] Succinct Labs, "SP1: The fastest zkvm," https://github.com/succinctlabs/sp1, 2024.

[4] A. Arun, S. Setty, and J. Thaler, "Jolt: SNARKs for virtual machines via lookups," in *Advances in Cryptology – EUROCRYPT 2024*, 2024.

[5] OpenVM Contributors, "OpenVM: A modular zkvm framework," https://github.com/openvm-org/openvm, 2024.

[6] Nexus Labs, "Nexus zkVM," https://github.com/nexus-xyz/nexus-zkvm, 2024.

[7] Brevis Network, "Pico: A modular and performant zkvm," https://github.com/brevis-network/pico, 2025, announcement: https://blog.brevis.network/2025/02/11/introducing-pico-a-modular-and-performant-zkvm/.

[8] Succinct Labs, "Optimism chooses succinct to bring ZK to the superchain," Succinct Labs Blog, 2026, sP1 zkVM secures over $5.5B TVS across 35+ customers. [Online]. Available: https://blog.succinct.xyz/optimism-superchain/

[9] J. Swihart, "Zcash counterfeiting vulnerability successfully remediated," Electric Coin Company, 2019. [Online]. Available: https://z.cash/zcash-counterfeiting-vulnerability-successfully-remediated/

[10] R. Hackett, "Zcash discloses vulnerability that could have allowed 'infinite counterfeit' cryptocurrency," Fortune, 2019. [Online]. Available: https://fortune.com/crypto/2019/02/05/zcash-vulnerability-cryptocurrency/

[11] H. Wen, J. Stephens, Y. Chen, K. Ferles, S. Pailoor, K. Charbonnet, I. Dillig, and Y. Feng, "Practical security analysis of zero-knowledge proof circuits," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/wen

[12] S. Pailoor, Y. Chen, F. Wang, C. Rodríguez-Núñez, J. Van Geffen, J. Morton, M. Chu, B. Gu, Y. Feng, and I. Dillig, "Automated detection of under-constrained circuits in zero-knowledge proofs," in *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, 2023.

[13] C. Hochrainer, V. Wüstholz, and M. Christakis, "Arguzz: Testing zkvms for soundness and completeness bugs," 2025. [Online]. Available: https://arxiv.org/abs/2509.10819

[14] RISC-V Software Source Authors, "riscv-tests: Unit tests for RISC-V processors," https://github.com/riscv-software-src/riscv-tests, accessed: 2026-05-06.

[15] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Fast Reed-Solomon interactive oracle proofs of proximity," in *45th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2018, pp. 14:1–14:17. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2018.14

[16] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *Advances in Cryptology – ASIACRYPT 2010*, 2010, pp. 177–194.

[17] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, no. 2–3, pp. 235–256, 2002.

[18] A. Waterman and K. Asanović, "The RISC-V instruction set manual, volume I: User-level ISA, document version 20191213," RISC-V Foundation, Tech. Rep., 2019. [Online]. Available: https://docs.riscv.org/reference/isa/v20191213/_attachments/riscv-unprivileged.pdf

[19] M. Zalewski, "American fuzzy lop (AFL)," https://lcamtuf.coredump.cx/afl/, 2014.

[20] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 2123–2138.

[21] Cantina, "SP1 audit report," Succinct Labs SP1 audits, 2024. [Online]. Available: https://github.com/succinctlabs/sp1/blob/main/audits/cantina.pdf

[22] KALOS, "SP1 recursion VM audit," Succinct Labs SP1 audits, 2024. [Online]. Available: https://github.com/succinctlabs/sp1/blob/main/audits/kalos.md

[23] G. r. Kim, "SP1 audit notes," Succinct Labs SP1 audits, 2024. [Online]. Available: https://github.com/succinctlabs/sp1/blob/main/audits/rkm0959.md

[24] Veridise, "SP1 audit report," Succinct Labs SP1 audits, 2024. [Online]. Available: https://github.com/succinctlabs/sp1/blob/main/audits/veridise.pdf

[25] Succinct Labs, "SP1 v4 audit notes," Succinct Labs SP1 audits, 2025. [Online]. Available: https://github.com/succinctlabs/sp1/blob/main/audits/sp1-v4.md

[26] Cantina, "OpenVM v1 audit report," OpenVM audits, 2025. [Online]. Available: https://github.com/openvm-org/openvm/blob/main/audits/v1-cantina-report.pdf

[27] OpenVM, "OpenVM v1 internal audit findings," OpenVM audits, 2025. [Online]. Available: https://github.com/openvm-org/openvm/tree/main/audits/v1-internal

[28] Hexens, "RISC Zero zkVM security audit," risc0/rz-security, 2023. [Online]. Available: https://github.com/risc0/rz-security/blob/main/audits/zkVM/hexens_zkVM_20231031.pdf

[29] Veridise, "RISC Zero zkVM security audit," risc0/rz-security, 2025. [Online]. Available: https://github.com/risc0/rz-security/blob/main/audits/zkVM/veridise_zkVM_20250224.pdf

[30] H. Takahashi, J. Kim, S. Jana, and J. Yang, "zkFuzz: Foundation and framework for effective fuzzing of zero-knowledge circuits," in *2026 IEEE Symposium on Security and Privacy (S&P)*, 2026, pp. 919–938. [Online]. Available: https://www.computer.org/csdl/proceedings-article/sp/2026/606500a901/2bojvL4Zswo

[31] J. Liu, I. Kretz, H. Liu, B. Tan, J. Wang, Y. Sun, L. Pearson, A. Miltner, I. Dillig, and Y. Feng, "Certifying zero-knowledge circuits with refinement types," in *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 2024, pp. 1741–1759. [Online]. Available: https://doi.org/10.1109/SP54263.2024.00078

[32] C. Hochrainer, A. Isychev, V. Wüstholz, and M. Christakis, "Fuzzing processing pipelines for zero-knowledge circuits," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2025.

[33] D. Xiao, Z. Liu, Y. Peng, and S. Wang, "MTZK: Testing and exploring bugs in zero-knowledge (ZK) compilers," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025.

[34] H. Wen, H. Liu, J. Song, Y. Chen, W. Guo, and Y. Feng, "FORAY: Towards effective attack synthesis against deep logical vulnerabilities in DeFi protocols," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

[35] J. Liu, J. Song, Y. Chen, H. Liu, H. Wen, L. Pearson, Y. Chen, and Y. Feng, "Tabby: A synthesis-aided compiler for high-performance zero-knowledge proof circuits," in *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2025.

[36] M. Wang, J. Liang, C. Zhou, Z. Wu, J. Fu, Z. Su, Q. Liao, B. Gu, B. Wu, and Y. Jiang, "Data coverage for guided fuzzing," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 2511–2526. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/wang-mingzhe

[37] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[38] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[39] LLVM Project, "LibFuzzer – a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, 2015.

[40] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for deep bugs with grammars," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[41] P. Srivastava and M. Payer, "Gramatron: Effective grammar-aware fuzzing," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[42] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOpt: Optimized mutation scheduling for fuzzers," in *Proceedings of the USENIX Security Symposium*, 2019.

[43] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *Proceedings of the USENIX Security Symposium*, 2020.

[44] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: Efficient domain-independent differential testing," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[45] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "SemFuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

APPENDIX A
FULL OBLIGATION ENUMERATION AND LINKED BENCHMARK ENTRIES

This appendix lists all 60 constraint obligations enumerated by BEAK, organized by the 10 functional groups of Table II. For each obligation we give its identifier, a short description, and the known benchmark entry it triggers on, drawn from the 37-bug known benchmark of Section VI-B. An ^{A} superscript marks a known benchmark entry also detected by ARGUZZ under the finite-time reproduction criterion used in RQ1; we use this letter rather than † to avoid collision with the ablation marker in Section VI-C. An empty entry indicates an obligation derived from specification analysis with no currently mapped known benchmark bug. A single benchmark bug may map to multiple obligations (e.g., `Jolt-Dory-ShortTrace-01` activates both `interaction.commit_pow2` and `row.trace_pow2`); the 31 linked entries below correspond to 26 unique benchmark bugs detected by BEAK.

Obligation	Description	Linked known benchmark entry
Decode (7)		
`decode.zero_reg`	x0 immutability: writes targeting `x0` must be discarded	
`decode.rd_decomp`	rd bit decomposition consistency over field encoding	Risc0-RangeCheck-Audit-r10
`decode.upper_imm`	upper-immediate (LUI/AUIPC) materialization into 32-bit value	Jolt-Immediate-01[A] OpenVM-RangeCheck-Audit-o7
`decode.operand_route`	operand routing from decoded fields to the execution pipeline	Risc0-TripleReg-01
`decode.prog_table_pow2`	program table 2^k capacity threshold	Jolt-HighBytecode-01
`decode.fmt_imm_reasm`	format-specific immediate reassembly and sign extension (I/S/B/U/J)	OpenVM-SignBit-Audit-o8
`decode.illegal_opcode`	illegal opcode trap	
ALU (8)		
`alu.imm_limb`	immediate byte-piece (limb) decomposition and recomposition	OpenVM-RangeCheck-Audit-o5
`alu.shift_mask`	shift-amount masking to low 5 bits	
`alu.cmp_bool`	SLT/SLTU comparison result is boolean	
`alu.add_carry`	ADD carry-chain consistency across limbs	
`alu.sub_borrow`	SUB borrow-chain consistency across limbs	
`alu.bitwise_lookup`	AND/OR/XOR lookup-table consistency	
`alu.cmp_aux`	comparison auxiliary witness consistency	
`alu.funct_dispatch`	funct3/funct7 dispatch to correct ALU operation	
Arithmetic (6)		
`arith.div_rem`	division remainder bounds ($0 \leq r < \lvert divisor \rvert$)	Jolt-DivRem-01 Risc0-Div-Audit-r9
`arith.signed_overflow`	signed overflow at -2^{31} edge cases	
`arith.mul_decomp`	multiplication product decomposition into hi/lo limbs	Nexus-MulCarry-01[A]
`arith.div_by_zero`	division-by-zero result convention ($q = 2^{32} - 1$, $r = dividend$)	
`arith.mulh_consistency`	MUL/MULH high-low consistency on the same operands	
`arith.signed_unsigned_mul`	signed/unsigned multiplication sign correction	Jolt-Mulhsu-01[A]
Control (8)		
`control.pc_limb`	PC byte-piece consistency across steps	
`control.ecall_arg`	ecall argument register encoding	Risc0-RangeCheck-Audit-r8 Pico-Ecall-OperandWord-01
`control.entrypoint_bind`	entrypoint PC binding at trace start	Jolt-EntryPc-Binding-01
`control.ecall_word`	ecall word validity (selector + argument range)	
`control.branch_cond`	branch condition selection (signed vs. unsigned)	
`control.branch_target`	branch target arithmetic (PC + sign-extended offset)	
`control.jalr_lsb`	JALR target LSB clearing	
`control.link_reg`	link register binding (rd $\leftarrow$ PC+4 on JAL/JALR)	SP1-Recursion-JumpBinding-01

Table IV. Full obligation enumeration (1/2): Decode, ALU, Arithmetic, and Control groups.

Obligation	Description	Linked known benchmark entry
Execution (7)		
`exec.operand_bind`	source/dest operand binding to register file	SP1-Recursion-LoadBinding-01 SP1-Recursion-Bneinc-UpperLimbs-01
`exec.opcode_select`	opcode selector exclusivity	OpenVM-Memory-Audit-o19 Pico-ReadWrite-OpcodeSelector-01
`exec.control_flow`	control-flow consistency between PC and executed instruction	SP1-Pc-Audit-s28[A]
`exec.mem_effect`	memory-effect binding to memory subsystem	
`exec.subword_write`	sub-word (byte/half) write boundary handling	
`exec.segment_pow2`	segment boundary 2^k threshold	Risc0-ControlDone-Cycle-01
`exec.bytecode_bind`	bytecode-to-operand binding (instruction word $\rightarrow$ decoded operands)	
Memory (12)		
`mem.addr_align`	address alignment for word/half accesses	SP1-Unaligned-Memory-01
`mem.ts_order`	timestamp ordering (monotonic per address)	
`mem.addr_route`	address-space routing (RAM vs. I/O vs. private memory)	OpenVM-AddrSpace-Audit-o51
`mem.sign_ext`	sign extension for LB/LH	
`mem.store_load_flow`	store-load value-flow consistency	Nexus-Operand-01
`mem.cap_pow2`	memory capacity 2^k threshold	Nexus-MemorySize-01 Jolt-RamSize-01
`mem.raw_consistency`	read-after-write consistency within a trace	
`mem.byte_offset_mask`	byte offset masking within a word	
`mem.addr_overflow`	address overflow at 2^{32} boundary	SP1-AddrSlice-Overflow-01
`mem.rw_direction`	read/write direction flag binding	SP1-Memory-Audit-s27
`mem.init_finalize`	memory init/finalize boundary rows	
`mem.bounds`	memory range bounds enforcement	OpenVM-RangeCheck-Audit-o25
Lookup (5)		
`lookup.boolean_mult`	boolean multiplicity in lookup arguments	OpenVM-Multiple-Audit-o15
`lookup.xor_mult`	XOR multiplicity in lookup arguments	OpenVM-Overflow-Audit-o1
`lookup.acc_pow2`	lookup accumulator 2^k threshold	
`lookup.range_pow2`	range-table 2^k threshold	
`lookup.table_pop`	lookup table population (every used row is materialized)	
Interaction (2)		
`interaction.digest`	cross-component digest routing	SP1-Digest-Audit-s29
`interaction.commit_pow2`	commitment vector 2^k threshold	Jolt-Dory-ShortTrace-01[A]
Row (2)		
`row.padding_leak`	padding row interaction leakage into live constraints	OpenVM-Timestamp-Audit-o3 SP1-Padding-Audit-s26
`row.trace_pow2`	trace padding 2^k threshold	Jolt-Dory-ShortTrace-01[A]
Time (3)		
`time.boundary_origin`	timestamp boundary origin consistency	OpenVM-Timestamp-Audit-o2 OpenVM-Timestamp-Audit-o26
`time.diff_range`	monotonic timestamp difference range	
`time.fp_wrap`	$\mathbb{F}_p$ modular wrap on timestamp arithmetic	

Table V. Full obligation enumeration (2/2): Execution, Memory, Lookup, Interaction, Row, and Time groups.