

Zorya: Automated Concolic Execution of Single-Threaded Go Binaries

Karolina Gorna

Telecom Paris and Ledger Donjon
Paris, France
karolina.gorna@telecom-paris.fr

Yannick Seurin

Ledger Donjon
Paris, France
yannick.seurin@ledger.com

Nicolas Iooss

Ledger Donjon
Zurich, Switzerland
nicolas.iooss@ledger.com

Rida Khatoun

Telecom Paris
Palaiseau, France
rida.khatoun@telecom-paris.fr

Abstract

Go's adoption in critical infrastructure intensifies the need for systematic vulnerability detection, yet existing symbolic execution tools struggle with Go binaries due to runtime complexity and scalability challenges. In this work, we build upon Zorya, a concolic execution framework that translates Go binaries to Ghidra's P-Code intermediate representation to address these challenges. We added the detection of bugs in concretely not taken paths and a multi-layer filtering mechanism to concentrate symbolic reasoning on panic-relevant paths. Evaluation on five Go vulnerabilities demonstrates that panic-reachability gating achieves 1.8–3.9× speedups when filtering 33–70% of branches, and that Zorya detects all panics while existing tools detect at most two. Function-mode analysis proved essential for complex programs, running roughly two orders of magnitude faster than starting from `main`. This work establishes that specialized concolic execution can achieve practical vulnerability detection in language ecosystems with runtime safety checks.

CCS Concepts

- Software and its engineering → Software verification and validation;
- Security and privacy → Software security engineering.

Keywords

Concolic execution, Go, Symbolic constraints, Predicate collection, Vulnerabilities detection, P-Code

ACM Reference Format:

Karolina Gorna, Nicolas Iooss, Yannick Seurin, and Rida Khatoun. 2026. Zorya: Automated Concolic Execution of Single-Threaded Go Binaries. In *The 41st ACM/SIGAPP Symposium on Applied Computing (SAC '26), March 23–27, 2026, Thessaloniki, Greece*. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3748522.3779940>



This work is licensed under a Creative Commons Attribution 4.0 International License.
SAC '26, Thessaloniki, Greece
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2294-3/2026/03
<https://doi.org/10.1145/3748522.3779940>

1 Introduction

Modern software systems increasingly rely on languages that enforce strong safety checks at runtime, yet failures still occur. Go language [11], now prevalent in cloud infrastructure and blockchain systems, enforces many memory-safety properties through runtime checks but still exhibits panics when safety invariants are violated and can even suffer segmentation faults in the presence of `unsafe`, `cgo`, or low-level runtime bugs [8, 28]. These panics are often input-dependent, triggered only when specific inputs cause nil-pointer dereferences, slice-bound violations, or nil-map operations for instance. This deterministic relationship between inputs and failures makes Go vulnerabilities well-suited to automated constraint solving. However, existing tools lack the language-specific modeling required to exploit this property.

Concolic execution, which couples concrete and symbolic execution, offers systematic vulnerability discovery by treating inputs as symbolic variables, executing programs to gather path predicates, then solving constraints to generate inputs exploring alternative paths [2]. However, existing concolic executors face fundamental challenges when applied to Go binaries: **C1: Go-Specific Runtime Complexity.** Go binaries embed a heavyweight runtime (garbage collection, stack management, interface dispatch, slice/map/string handling) and Go-specific runtime calls and syscalls that general-purpose symbolic executors for C/C++ do not model [26]. **C2: Scalability Without Precision Loss.** Analyzing real-world Go programs requires managing vast symbolic state spaces while maintaining detection accuracy, a tension that traditional symbolic executors resolve by sacrificing either coverage or soundness.

This paper extends Zorya [14], a binary-level concolic execution framework that lifts binaries to Ghidra [17] P-Code. The central addition is an instantiation of classical target-directed reachability for Go panic functions, which we term *panic-gated exploration*. For each concrete starting state, Zorya restricts symbolic reasoning to branches that may reach known panic locations, using a conservative backward reachability pre-analysis over the control-flow graph instead of exploring all execution paths. This specialization of reachability to Go's panic mechanism, combined with additional filtering mechanisms, lays the foundation for scalable concolic analysis of large Go binaries and, by extension, other compiled languages.

Assumptions. Zorya assumes correct Ghidra disassembly; execution halts if jumps target unidentified code. We mitigate this via

preprocessing and compiler predictable layouts. Currently, Zorya analyzes non-interactive binaries requiring inputs at initialization.

Contributions:

- **Panic-gated concolic execution:** A filtering cascade addressing Go-specific complexity (C1) and scalability (C2) through precomputed reachability, constraint context filtering, and AST-based pre-checking, reducing SMT solver queries by two orders of magnitude. Zorya is open source at <https://github.com/Ledger-Donjon/zorya>.
- **Empirical validation:** Evaluation on five Go vulnerabilities demonstrates 1.2–3.9× speedups via optimization, with 5/5 detection versus 0–2/5 for existing tools. Evaluation results can be found at <https://github.com/Ledger-Donjon/zorya-evaluation>.
- **Go vulnerability corpus:** Initial dataset enabling reproducible research at https://github.com/Ledger-Donjon/logic_bombs_go.

2 Background

2.1 P-Code Intermediate Representation

Ghidra’s P-Code is a platform-independent intermediate language that normalizes diverse instruction sets into a consistent representation. Each native instruction decomposes into one or more P-Code operations (e.g., LOAD, STORE, INT_ADD, CBranch). P-Code uses varnodes to represent storage locations: registers, memory addresses, constants, and temporary values [16].

Ghidra distinguishes between raw P-Code, which directly translates machine instructions preserving all low-level details and temporary registers, and high-level P-Code, which applies simplifications for decompilation [7]. Zorya uses raw P-Code to maintain instruction-level precision required for concolic execution. This normalization facilitates reusing the same concolic infrastructure across x86-64, ARM, and other architectures, while only requiring architecture-specific P-Code lifters and syscall models rather than a full reimplementation of instruction semantics per platform.

2.2 Symbolic and Concolic Execution

Symbolic execution analyzes programs by representing inputs as symbolic variables instead of fixed values. When the program reaches a conditional branch, a symbolic executor uses an SMT (Satisfiability Modulo Theories) solver to check if alternative execution paths are possible. SMT solvers such as Z3 [6] handle constraints involving integers, bit-vectors, arrays, and other data types, generating concrete input values that satisfy logical conditions. However, symbolic execution faces a fundamental scalability problem: path explosion. Each conditional branch can double the number of paths to analyze, quickly making complete program analysis impractical.

Concolic execution, combining concrete and symbolic execution, mitigates this problem. This approach maintains two parallel states: a concrete state that executes the program with real values, and a symbolic state that records logical constraints on inputs [22]. The concrete execution selects a specific path through the program based on the binary’s inputs, while the symbolic state captures conditions needed to reach alternative paths. This hybrid strategy improves scalability by following one concrete path at a time while

using symbolic reasoning to discover inputs that trigger different behaviors.

2.3 Go language and compilers

2.3.1 Go components. Go, introduced in 2009, is a statically typed, compiled language designed for simple concurrent programming [11]. Concurrency is a core feature: lightweight goroutines are started with the go keyword and communicate primarily through typed channels, while also supporting shared-memory synchronization when needed [10]. The language provides structured types such as slices and maps; slices are descriptors storing a pointer to the underlying array, a length, and a capacity, and out-of-bounds accesses trigger runtime panics due to mandatory bounds checking [13]. Go inserts runtime checks to prevent many invalid memory accesses, but it is not fully memory safe: the unsafe package, cgo, and low-level runtime code can still cause segmentation faults and other memory-safety violations [12]. We therefore focus first on detecting calls to the panic mechanism, which captures most well-structured failures in Go and TinyGo binaries.

2.3.2 Go compiler. The standard Go toolchain produces statically linked binaries that bundle the runtime, including the scheduler, garbage collector (GC), and type information, into each executable [11]. At startup, execution enters the runtime, which initializes the scheduler, GC, and global state before invoking the user-defined main function, interleaving user code with runtime services throughout the binary. The compiler emits DWARF debug information describing variable locations, types, and source mappings, which we use to reconstruct function arguments and high-level types in our binary-level analysis.

2.3.3 TinyGo compiler. TinyGo is an alternative Go toolchain for microcontrollers and other resource-constrained platforms [25]. It introduces a new compiler, which uses (mostly) the standard Go libraries and LLVM to generate machine code, and a lightweight runtime that implements a memory allocator, scheduler, string operations, and partially re-implemented packages such as sync and reflect. This design avoids limitations of the standard toolchain on microcontrollers, including missing support for instruction sets such as Thumb and AVR and large runtime footprints [25]. By default, TinyGo executes Go code on a single operating system thread and multiplexes goroutines in user space, which removes many low-level thread interleavings and makes TinyGo binaries a simpler first target for symbolic and concolic analysis. Many safety checks, such as slice bounds and nil dereferences, still lead to panics, while TinyGo’s documentation notes that stack overflows and low-level bugs can still cause segmentation faults and hard crashes [25]. Our prototype therefore targets panic-induced failures in TinyGo binaries, leaving non-panic crashes such as segmentation faults to future work.

3 Motivating Example

We illustrate Zorya’s capabilities on a TinyGo-compiled calculator that accepts two operands and an operator. The aim of using the TinyGo compiler is to work with single-threaded binaries, easier to analyze symbolically, as a first approach. This example program (Listing 1) is correct on typical inputs (e.g., 2 + 3), but contains an

injected, state-dependent bug: when both operands equal 5, a nil-pointer dereference causes a runtime panic (line 10). Using Zorya’s binary-argument symbolic exploration, command-line arguments are modeled symbolically at the starting address, and execution proceeds concolically over P-Code, maintaining both concrete values and symbolic expressions, while building the symbolic path predicate. Upon encountering relevant control-flow splits, Zorya queries an SMT solver to synthesize inputs that satisfy the path conditions leading to the panic site; in this case, it recovers the precise trigger `operand1 == 5 && operand2 == 5`.

```

1 func coreEngine(num1 int, operator string, num2
2     int) (int, error) {
3     var result int
4     switch operator {
5         case "+": result = num1 + num2
6         case "-": result = num1 - num2
7         case "*": result = num1 * num2
8         [...]
9         // Intentional panic trigger
10        if num1 == 5 && num2 == 5 {
11            var p *int; *p = 0
12        }
13    return result, nil
}
```

Listing 1: Core function with injected panic.

4 Concolic Execution and Path Predicate Collection

Zorya extracts solver-ready predicates from conditional flags at each P-Code conditional branching (CBranch instruction). Let Π denote the *path predicate*, the conjunction of branch conditions along the concrete path, and ϕ the symbolic predicate from the branch flag. On traversing an edge, Zorya updates $\Pi' = \Pi \cup \{\phi\}$, accumulating only execution-induced constraints. Concrete values guide the primary path while symbolic expressions enable alternative branch exploration.

Predicates are SMT booleans and bit-vectors, with bit-vectors interpreted as true if and only if non-zero. Negation follows standard semantics. For example, if a guard simplifies to the SMT-LIB expression (`= len!142 #x01`), Zorya asserts the symbolic length equals 1. This instruction-driven accumulation preserves concrete-run fidelity while producing precise, solver-suitable constraints.

5 Negated-path Exploration

As illustrated in the Figure 1, Zorya performs targeted *negated-path exploration* to synthesize vulnerability-triggering inputs through a panic-gated approach: only branches toward known panic addresses \mathcal{P} undergo symbolic exploration. At each CBranch with target t , Zorya applies $\text{IsPanicXref}(t, \mathcal{P})$. For gated branches with predicate ϕ , the framework snapshots solver state (push), asserts $\Pi' = \Pi \cup \{\neg\phi\}$, queries satisfiability, extracts models on SAT, and discards on UNSAT (pop). Exploration proceeds when ϕ references symbolic arguments or when prior constraints exist, capturing transitive symbolic dependencies.

To achieve scalability, Zorya implements cascaded filtering that reduces solver invocations through four complementary optimizations applied sequentially at each CBranch. Each stage refines the candidate set, with later stages performing more expensive analysis on progressively smaller subsets:

1./ Precomputed Panic Reachability (coarse filter). Zorya performs static backward analysis from panic-fatal-abort sites to construct \mathcal{R} , the set of panic-reachable basic blocks. The algorithm starts from known panic function addresses (identified via symbol analysis of `runtime.panic*`, `runtime.fatal*`, etc.) and traverses the control-flow graph backwards, marking all blocks that can reach these sites through any path. This includes both direct calls and indirect control flow through function pointers or virtual dispatch. The analysis is conservative: if uncertainty exists about reachability, blocks are marked reachable, ensuring soundness. At runtime, branches where $pc \notin \mathcal{R}$ and $t \notin (\mathcal{R} \cup \mathcal{P})$ are immediately discarded without symbolic analysis.

2./ Internal Target Detection (artifact filter). P-Code translation generates internal branches targeting other P-Code statements in the same assembly instruction. Contrary to usual branches, Ghidra represents these targets in the `const` space, enabling Zorya to detect and exclude them from symbolic analysis. These artifacts arise from complex instruction semantics (e.g., x86 conditional moves) that decompile into conditional P-Code micro-operations never representing user-level control flow.

3./ Constraint Context Filtering (relevance filter). Exploration requires either (i) ϕ syntactically references tracked arguments, or (ii) non-empty solver constraints from prior branches, focusing on symbolically-relevant paths. This filter recognizes that branches independent of symbolic inputs cannot contribute to vulnerability discovery: if a branch depends only on constants or non-symbolic state, no input modification can alter its behavior.

4./ AST Pre-Check (speculative filter). Before SMT invocation, lightweight AST traversal from `addralt` searches for panic calls within bounded depth (10 blocks). Only `FOUND_PANIC` results proceed to solving. This speculative exploration executes symbolically without solver queries, simulating execution along the negated path to determine whether a panic is reachable. The bounded depth prevents unbounded exploration while capturing panic calls within typical inlined function bodies.

Single-Evaluation Commitment (efficiency guarantee). Upon AST confirmation, exactly one solver query executes, minimizing overhead via deferred SMT until high-confidence targets emerge. Traditional symbolic execution may query the solver at every branch; Zorya’s deferred approach batches symbolic reasoning until panic proximity is confirmed, concentrating solver invocations on high-value targets.

These optimizations reduce solver queries by up to two orders of magnitude (Section 8). Conceptually, panic-gated exploration instantiates classical target-directed reachability: panic sites play the role of assertions, and the precomputed panic-reachability set over-approximates the basic blocks that can flow to these targets. In contrast to Backward-Bounded DSE (BB-DSE), which uses backward symbolic reasoning to answer *infeasibility* questions on obfuscated x86 code [1], Zorya combines a lightweight backward control-flow reachability analysis with forward concolic execution to answer

feasibility questions for panic sites in Go/TinyGo binaries, guided by Go-specific modeling and the multi-layer filter cascade.

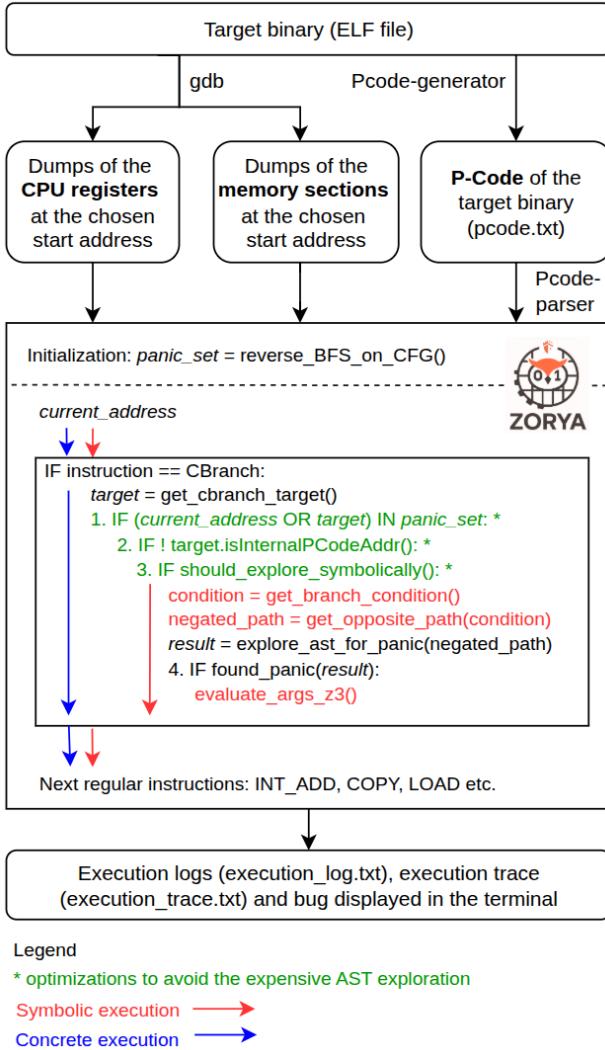


Figure 1: Overview of Zorya workflow, including the optimizations of the negated-path exploration

6 Algorithm

6.1 Structure

Algorithm 1 implements Zorya's optimized concolic execution through three phases. The **initialization phase** (lines 2-11) establishes the analysis infrastructure: lines 2-4 create the Z3 context and solver, while lines 5-7 extract essential metadata from the binary (function signatures via DWARF, panic addresses via symbol analysis, and the precomputed panic-reachability set \mathcal{R}). Lines 8-11 initialize symbolic arguments by creating symbolic variables for each function parameter and writing them to their designated locations in the CPU state Σ .

The **main execution loop** (lines 12-32) processes P-Code instructions sequentially. For each instruction, the algorithm first checks if it represents a conditional branch (line 16). Non-branch instructions proceed directly to execution (line 34). For conditional branches, the four-layer optimization cascade applies: line 17 implements *internal target detection*, filtering out P-Code internal addresses; line 18 applies *reachability gating*, discarding branches where neither the current location pc nor target t are panic-reachable; line 21 enforces *constraint context filtering*, requiring that the branch condition ϕ references symbolic arguments or that prior constraints exist; and line 22 executes *AST pre-checking*, speculatively exploring the negated path to confirm panic reachability before expensive SMT invocation.

The **SMT solving phase** (lines 23-27) executes only when all filters pass. The solver snapshots its state (line 23), asserts the negated condition $\neg\phi$ (line 24), and checks satisfiability (line 25). On SAT, the algorithm returns a model—concrete input values that would follow the negated path to the panic. On UNSAT, the solver restores its previous state (line 27) and continues execution. This structure implements *single-evaluation commitment*: exactly one solver query per confirmed panic-relevant branch.

Algorithm 1 Zorya: Optimized Concolic Execution

```

1: procedure ZORYA_CONCOLIC(binary, start_addr)
2:    $C \leftarrow Z3Context()$ 
3:    $S \leftarrow Z3Solver(C)$ 
4:    $\Sigma \leftarrow InitializeCPUState(C)$ 
5:    $\mathcal{F} \leftarrow ExtractFunctionSigs(binary)$ 
6:    $\mathcal{P} \leftarrow LoadPanicAddresses(binary)$ 
7:    $\mathcal{R} \leftarrow PrecomputePanicReach(\mathcal{P})$ 
8:   for arg  $\in \mathcal{F}.\text{target.arguments} do
9:      $\alpha \leftarrow CreateSymbolicVar(arg, C)$ 
10:    WriteToLocation(arg.reg,  $\alpha$ ,  $\Sigma$ )
11:   end for
12:    $pc \leftarrow start\_addr$ 
13:   while  $pc \neq \perp$  do
14:      $inst \leftarrow FetchInstruction(pc)$ 
15:      $t \leftarrow TargetOf(inst)$ 
16:     if inst == CBranch then
17:       if  $\neg\text{isInternalPCodeAddr}(t)$  then                                 $\triangleright$  Skip sub-instr
18:         if  $pc \in \mathcal{R} \vee t \in (\mathcal{R} \cup \mathcal{P})$  then
19:            $\phi \leftarrow ExtractCondition(inst)$ 
20:            $addr_{alt} \leftarrow NegatedPath(inst, pc)$ 
21:           if  $\phi$  refs  $\mathcal{F}.\text{args} \vee S$  has constraints then
22:             if ExploreAST( $addr_{alt}$ ) == PANIC then
23:                $S.push()$ 
24:                $S.assert(\neg\phi)$ 
25:             if  $S.\text{check}() = \text{SAT}$  then
26:               return GetModel( $S$ )
27:             end if
28:              $S.pop()$ 
29:           end if
30:         end if
31:       end if
32:     end if
33:   end if
34:    $(\Sigma, pc) \leftarrow Execute(inst, \Sigma, pc)$ 
35:   end while
36: end procedure$ 
```

6.2 Correctness, Soundness and Completeness

Zorya is *correct along explored paths*: for each concrete execution it follows, the symbolic state is kept consistent with the concrete state, so the path predicate Π faithfully encodes encountered branch conditions. It is *sound with respect to path feasibility* under ideal assumptions on Z3, P-Code semantics, DWARF information, the memory model, and absence of unmodeled nondeterminism: any model M satisfying $\Pi \cup \{\neg\phi\}$ should then drive execution along the corresponding negated path to the targeted panic. In practice, engineering choices such as *lazy concretization* of symbolic values used in memory accesses, bounded materialization of slices/strings, and pointer anchoring introduce approximations that can miss feasible paths or admit spurious ones in corner cases. As a result, Zorya is *boundedly complete*: these heuristics and the panic-gated search strategy improve scalability at the expense of full soundness over all possible executions.

7 Implementation

Zorya's implementation addresses three key challenges: bridging P-Code to executable semantics, reconstructing Go's data layout at binary level, and managing SMT solver efficiency.

P-Code Interpretation. The Rust engine maps varnodes to paired concrete/symbolic values, with on-demand conversion for registers, memory, and constants. A critical challenge is P-Code's temporary registers, which persist across instructions but lack symbolic identity; Zorya tracks these via instruction handling.

Go Data Reconstruction. Without source types, Zorya recovers structures through DWARF and runtime inspection. Slices materialize as $(\text{ptr}, \text{len}, \text{cap})$ triples where pointers are *anchored* to concrete addresses ($\text{ptr} == 0xNNNN$) to prevent symbolic pointer arithmetic bottlenecks, while length/capacity remain symbolic within bounds $[0, 64]$. Strings follow similar patterns (256-byte limit). Zorya relies on DWARF debug information to extract function signatures and argument locations; binaries compiled without debug symbols are not currently supported.

Z3 Optimize Solver. Zorya uses Z3's Optimize solver to handle symbolic constraints. For symbolic slices, Zorya enforces hard constraints on length and capacity ($0 \leq \text{len} \leq 64$) to maintain tractability while permitting the solver to explore feasible values within these bounds. Push/pop scoping maintains solver state across negated-path exploration queries.

Concolic Synchronization. Operations like array/map indexing with symbolic values ($\text{slice}[\text{symbolic_index}]$) require concrete evaluation of `symbolic_index` for memory access while retaining symbolic expressions for constraint solving. Zorya employs "lazy concretization": when a symbolic value must be used concretely (e.g., as a memory offset), the framework simplifies the symbolic expression and evaluates it under the current model to obtain a concrete value, while preserving the original symbolic expression for later SMT queries. This approach is critical for Go's runtime, which performs extensive bounds checking with nested conditional expressions that would lead to exponential symbolic expression growth under pure symbolic propagation.

8 Evaluation

We evaluate Zorya's panic-gated optimizations on theoretical and real-world Go binaries, measuring detection accuracy, optimization impact, and performance scaling, under the following research questions:

- **RQ1:** How effectively does Zorya detect vulnerabilities in Go binaries compared to existing symbolic execution tools?
- **RQ2:** What performance improvements do panic-gating and multi-layer filtering provide across different vulnerability types?
- **RQ3:** How does starting point selection (main vs. function) interact with optimization effectiveness?

8.1 Experimental Setup

Experiments ran on 64-bit Linux with Intel Core i7-1165G7 (4 cores/8 threads, 2.8 GHz base, 4.7 GHz boost) and 32 GB RAM, using Zorya v0.0.4, Ghidra v11.1.4, and TinyGo v0.33.0. We measured average detection time over 5 runs per seed input, comparing unoptimized Zorya (all CBranch explored) against optimized Zorya (multi-layer filtering enabled). We evaluated against BINSEC v0.10.1 [3], MIASIM v0.1.3 [4], radius2 v1.0.16 [21], and Owi (verison from on Aug 29, 2025) [18] with default configurations.

8.2 Benchmark

Our suite combines theoretical programs isolating specific runtime failures with a real-world vulnerability from production audits. The structure of this benchmark is inspired by Logic Bomb [30] work with focused Go programs:

Theoretical programs.

- `crashme` (nil map assignment): conditional dereference after byte check, stressing pointer reasoning.
- `invalid-shift` (buffer overflow): byte-as-index into fixed array, exercising ASCII constraint reasoning.
- `panic-index` (index out-of-bounds): numeric argument guarding slice access, isolating bounds-check logic.
- `broken-calculator` (nil dereference): arithmetic operation with nested control flow triggering dereference on specific operand pairs.

Real-world case: Omni Network. `omni-vuln` reproduces "OMP-12: Index Out Of Bounds Panic in GetMultiProof()" from Omni Network's audit [24]. Malformed Merkle trees with single-child internal nodes cause sibling index `s` to exceed `len(tree)-1`, triggering runtime panic on `tree[s]` access.

All binaries are single-threaded TinyGo compilations. We package these binaries and their triggering/non-triggering inputs as a Go vulnerability corpus called `logic_bombs_go`, released as an accompanying artefact at https://github.com/Ledger-Donjon/logic_bombs_go, to support reproducible evaluation and future work on Go binary analysis.

8.3 Results and Analysis

Table 1 shows that panic-gating optimizations provide complexity-dependent speedups. The "NA" (Not Adapted) designation indicates tools currently lack the CPU instruction and syscall support

required to analyze Go binaries, reflecting engineering priorities rather than fundamental limitations.

RQ1 - Detection Effectiveness.

Zorya detected all five vulnerabilities. Among comparison tools, BINSEC, a well-established binary-level symbolic executor with proven effectiveness on C/C++ programs, possesses sufficient x86-64 instruction coverage to attempt Go binary analysis. It detected `crashme` and `invalid-shift` in 1 second each, demonstrating efficient performance on programs with shallow control flow. BINSEC did not complete analysis for the three more complex cases (`panic-index`, `broken-calculator`, `omni-vuln`), which involve Go's runtime structures and Go-specific syscalls.

MIASM, radius2, and Owi are marked "NA" as they currently lack prerequisite support for Go binaries. These are capable tools in their target domains; MIASM and radius2's intermediate languages (IL and ESIL) were designed primarily for other architectures and do not yet model the specific x86-64 instruction sequences and system call patterns emitted by Go compilers. Owi was designed for WebAssembly symbolic interpretation and operates on a different instruction set architecture than native x86-64 Go binaries. Extending these tools to support Go would require architectural additions—instruction semantic rules, syscall handlers, and calling convention models, representing reasonable engineering investments that maintainers may pursue based on community needs.

Finding 1: Zorya detected 5/5 vulnerabilities through Go-specific modeling. BINSEC detected 2/5 simpler cases; MIASM/radius2/Owi do not yet support Go binary analysis (lacking CPU instruction and syscall models), highlighting the value of language-aware binary analysis.

RQ2 - Optimization Impact.

Multi-layer filtering provides speedups that vary significantly based on whether panic-reachability gating filters branches. When gating statistics show [0/N], meaning zero branches filtered—results are mixed: `crashme` improves from 45.1s to 35.6s (1.3 \times speedup) in main mode but degrades from 23.6s to 35.8s (0.7 \times slowdown) in function mode, while `invalid-shift` improves 1.5 \times (main) and 1.4 \times (function). These programs have shallow control flow where every branch is panic-reachable, so only the four non-gating optimizations apply. The inconsistent results suggest that optimization overhead (tracking gating statistics, AST pre-checks) can outweigh benefits when no branches are actually filtered.

When panic-reachability gating actively filters branches, results are consistently positive. `panic-index` filters 33–47% of branches and achieves 2.4–3.9 \times speedups; `broken-calculator` filters 35–70% and gains 1.8–2.2 \times ; `omni-vuln` filters 38% and improves 3.3 \times . Here, the gating optimization eliminates entire branches before other stages process them. For example, `panic-index` in function mode gates 8 of 17 branches (47%) and achieves 3.9 \times speedup, while main mode gates only 19 of 58 branches (33%) and achieves 2.4 \times speedup—suggesting that gating effectiveness matters more than the absolute number of branches filtered.

Notably, `omni-vuln` transforms from intractable (>7200s) to practical (83.1s) despite filtering only 38% of branches, demonstrating that even moderate filtering can enable analysis of complex real-world programs. The data reveals a threshold effect: programs where gating filters zero branches see marginal or negative results

due to the additional calculation of the reverse BFS (0.7–1.5 \times), while programs where gating filters any branches see substantial gains (1.8–3.9 \times). This confirms that panic-reachability gating is the critical optimization: when it activates, the four-layer cascade delivers strong performance. When it cannot filter branches, the remaining optimizations provide limited benefit or even introduce overhead.

Finding 2: Optimizations provide 1.8–3.9 \times speedups when panic-reachability gating filters branches (33–70% filtering). When no branches are filtered, results are inconsistent (0.7–1.5 \times), with one case showing slowdown, indicating that gating effectiveness is critical for optimization benefit.

RQ3 - Starting Point and Optimization Interaction.

Function mode benefits more from optimizations than main mode. For `panic-index`, function mode achieves 3.9 \times speedup compared to main mode's 2.4 \times . This reflects differing baseline costs: main mode includes Go runtime initialization overhead orthogonal to panic reachability, which moderates optimization impact. Function mode bypasses this infrastructure, concentrating on user logic where panic-gating more directly reduces exploration overhead.

Simpler programs (`crashme`, `invalid-shift`) show convergence after optimization (35–41s), suggesting both modes reach similar bottlenecks once filtering addresses shared overhead. The real-world `omni-vuln` illustrates a practical consideration: main mode remains intractable (>7200s) due to CLI parsing complexity, while function mode becomes viable (83.1s).

This difference of at least 87 \times indicates that function-level analysis may be preferable for certain deployment scenarios. This divergence reveals a natural trade-off in concolic execution design: main-mode analysis offers broader coverage by capturing argument parsing logic and environment interactions, but experiences path explosion in framework code (CLI parsers, logging initializers) unrelated to core vulnerability logic. Function-mode analysis trades some of this coverage for improved tractability, assuming developers can identify candidate vulnerable functions, a reasonable assumption for targeted security audits, though potentially less suitable for whole-program bug discovery. This suggests that a hybrid approach, where lightweight static analysis first identifies high-risk functions for deeper function-mode exploration, could combine the benefits of both strategies.

Finding 3: Function-mode analysis amplifies optimization benefits, achieving 1.8–3.9 \times speedups by concentrating filtering on user logic. Main mode's initialization overhead makes it less tractable for complex real-world cases.

8.4 Discussion

The correlation between gating statistics and speedup (0% gating/1.3 \times to 70% gating/3.9 \times) provides empirical support for multi-layer filtering. Programs with deep control flow benefit most from panic-reach precomputation, which prunes 35–70% of branches before symbolic analysis. Programs with shallow control flow see more modest gains, as expected when all branches are panic-relevant.

BINSEC's success on simpler cases demonstrates that general-purpose symbolic executors with comprehensive x86-64 support can effectively analyze Go binaries when execution remains relatively shallow. The challenges observed with more complex cases

Table 1: Evaluation of Zorya and other tools on theoretical and real-world vulnerabilities.

Binary (Type)	Size (KB)	Tested inputs (no panic)	Vuln. input(s)	Starting point	Zorya before opt. (sec)	Zorya after opt. (sec)	Gated/ Total	BINSEC (sec)	Other tools
crashme (Nil Map)	866	"a"; "B"; "100"	"K"	main.main crash()	45.1 23.6	35.6 35.8	0/4 0/1	1	NA
invalid-shift (Buffer overflow)	866	"10"; "42"; "1000"	"@"; "H?"; "d???"	main.main shift()	53.5 55.9	36.2 40.6	0/5 0/2	1	NA
panic-index (Index OOB)	885	"0"; "1"; "2"	arg1 > 3	main.main index()	489.9 148.3	203.1 38.3	19/58 8/17	NA	NA
broken-calc (Nil Dereference)	930	"2+3"; "5+1"; "6-5"	arg1=5 arg3=5	main.main coreEngine()	726.6 77.6	331.1 44.1	34/97 7/10	NA	NA
omni-vuln (Merkle tree)	1403	"a b c [e f [g h]]" -indices [1 3 5]"	"0 0 -indices 1"	main.main GetMultiProof()	>7200 273.1	>7200 83.1	- 12/32	NA	NA

Abbreviations: OOB = Out-of-Bounds; Before/After opt. = Before/After adding the optimizations to Zorya; Gated/Total = number of CBranch instructions gated by panic reachability out of total; Other tools = MIASM, radius2, Owi; NA: Not Adapted for Go binaries due to lack of support; Note: Average detection times are calculated with the inputs that do not trigger panics, over five runs each.

suggest that Go’s compiled runtime introduces semantic layers that may benefit from language-specific modeling. Tools marked “NA” do not yet include the foundational instruction and syscall support needed for Go analysis, a gap that could be addressed through targeted extensions if tool maintainers identify sufficient demand from their user communities. Zorya addresses these requirements through P-Code normalization and Go-aware symbolic types, offering one approach to binary-level Go vulnerability discovery that may complement existing tools’ strengths in other domains.

9 Limits and Improvements

Zorya has several areas for future enhancement. Currently, Zorya operates on non-interactive binaries, requiring all inputs at program initialization (command-line arguments or function parameters). This design choice simplifies the initial implementation but limits analysis of programs with runtime user interaction, file I/O loops, or network-driven input processing. Extending Zorya to handle incremental symbolic input during execution would broaden its applicability to interactive applications such as servers, REPLs, and streaming processors.

The panic-reachability analysis could be enhanced to eliminate the AST pre-check phase. Currently, the reverse BFS produces a simple set enabling constant-time membership queries. Augmenting this set with path information during preprocessing could avoid runtime AST exploration, but risks complicating the data structure and degrading lookup performance. We maintain the current two-phase design (reverse BFS + AST pre-check) to preserve linear-time panic-reachability queries, deferring finer-grained analysis to runtime when high-confidence targets emerge.

Additional improvements could include integrating fuzzing techniques to reduce reliance on concrete seed inputs for initial path exploration, dynamically adjusting AST exploration depth based on program complexity, and reconsidering the exclusion of internal Go functions from negated-path exploration to potentially capture runtime vulnerabilities.

Extending Zorya to multi-threaded Go binaries represents a significant research challenge. Go’s concurrency model, based on goroutines and channels, introduces interleaving explosion: each synchronization point multiplies the exploration space exponentially. Concurrency-specific panics, such as sending on closed channels, unsynchronized map access, or deadlocks, require symbolically modeling Go’s runtime scheduler and synchronization primitives. Despite these challenges, supporting concurrent programs would significantly expand Zorya’s applicability, as goroutines pervade real-world Go applications where concurrency bugs often manifest as panics under race conditions.

10 Related Work

Binary-level symbolic execution tools have trouble with Go binaries because of the Go runtime and data layout. LLVM IR-based tools such as Haybale [19] and SymSan [20] depend on gollvm [9], which is not widely used in production and does not yet support many real Go builds. Even when IR is available, Go features (slices as $\langle \text{ptr}, \text{len}, \text{cap} \rangle$, interfaces as $\langle \text{type}, \text{value} \rangle$, channels) are lowered to generic memory operations, which hides the checks and invariants needed for precise constraints. MAAT [27] also uses Ghidra’s P-Code like Zorya, but lacks Go-specific semantic modeling and cannot direct solving toward panic-relevant paths. In our evaluation, MAAT reported unsupported instruction errors on Go binaries, preventing successful analysis.

General-purpose binary analyzers such as Angr [29] and MIASM need Go-specific lifters, syscall models, and runtime stubs; without them they struggle with garbage collection and calling conventions. Radius2, built on radare2’s ESIL, uses a lightweight stack-based IR for emulation and analysis. ESIL does GhiHorn [5] translates P-Code to Horn clauses for proofs, which is hard to scale to large Go programs and is not focused on input generation. Owi targets WebAssembly and does not analyze native x86-64 Go binaries. BINSEC is first and foremost a general binary analysis framework (disassembly, taint, abstract interpretation), which also includes a symbolic execution engine. It provides strong x86-64 coverage and efficient

path exploration on binaries targeting C-like ABIs. However, it does not model Go's runtime (slices, interfaces, channels, panic routines) or Go-specific syscalls.

Targeted symbolic execution and line/target reachability have been studied extensively. Backward-Bounded DSE (BB-DSE) [1] performs backward symbolic reasoning from a chosen target to answer *infeasibility* questions on obfuscated x86 code (e.g., proving that a branch is dead). Zorya's panic-gated exploration is complementary: it also focuses on specific targets, but uses conservative backward control-flow reachability followed by forward concolic execution to answer *feasibility* questions for panic sites in Go/TinyGo binaries.

Go-focused work is limited. ColorGo [15] proposes directed concolic execution at the source level, but there is no public tool to compare against and it does not address binary-only settings. Duck-EGO [23] supports only basic types (e.g., ints, bools, simple maps) and does not handle strings, slices, structs, goroutines, or external libraries, which are common in real programs. Zorya addresses these gaps through a unique combination of P-Code-based analysis, DWARF-based argument recovery, binary-level Go type modeling, and panic-gated exploration, enabling practical vulnerability detection in compiled Go binaries without source code.

11 Conclusion

This paper presented several improvements to Zorya, a panic-gated concolic execution framework for Go binaries that combines P-Code lifting with Go-aware symbolic modeling. Evaluation on five vulnerabilities demonstrates that multi-layer filtering provides 1.8–3.9× speedups when panic-reachability gating filters 33–70% of branches, but introduces overhead when no filtering occurs, validating panic-gating as the critical optimization. Zorya detected all five vulnerabilities while comparison tools detected at most two, highlighting the value of language-specific binary analysis.

Function-mode analysis proved essential for real-world deployment, running roughly two orders of magnitude faster than main-mode on complex programs by bypassing runtime initialization overhead. Future work will extend Zorya to multi-threaded binaries and support incremental symbolic inputs. This work establishes that specialized concolic execution can achieve practical vulnerability detection in language ecosystems with runtime safety checks.

Acknowledgment

The authors would like to thank our anonymous reviewers for their valuable feedback. This research would not have been possible without the support of the Ledger Donjon team and the Telecom Paris INFRES department. We extend special thanks to Dr. Robin David for his invaluable guidance and advice throughout this work.

References

- [1] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA, 633–651. doi:10.1109/SP.2017.36
- [2] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90. doi:10.1145/2408776.2408795
- [3] CEA. 2025. binsec/binsec: BINSEC binary-level open-source platform. <https://github.com/binsec/binsec>
- [4] CEA-Sec. 2025. cea-sec/miasm. <https://github.com/cea-sec/miasm> original-date: 2014-10-28T15:05:47Z.
- [5] CERT Coordination Center. 2021. GhidHorn: Path Analysis in Ghidra Using SMT Solvers. <https://insights.sei.cmu.edu/blog/ghihorn-path-analysis-in-ghidra-using-smt-solvers/>
- [6] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24
- [7] Chris Eagle and Kara Nance. 2020. *The Ghidra Book: The Definitive Guide*. No Starch Press. Google-Books-ID: RVz6DwAAQBAJ.
- [8] Shuhao Fu and Yong Liao. 2024. Golang Defect Detection based on Value Flow Analysis. In *2024 9th International Conference on Electronic Technology and Information Science (ICETIS)*. ICETIS, 358–363. doi:10.1109/ICETIS61828.2024.10593695
- [9] Go-Community. 2017. gollvm: an LLVM-based Go compiler. <https://go.googlesource.com/gollvm/>
- [10] Google. 2025. Effective Go - The Go Programming Language. https://go.dev/doc/effective_go
- [11] Google. 2025. The Go Programming Language. <https://go.dev/>
- [12] Google. 2025. The Go Programming Language Specification - The Go Programming Language. <https://go.dev/ref/spec>
- [13] Google. 2025. Go Slices: usage and internals - The Go Programming Language. <https://go.dev/blog/slices-intro>
- [14] Karolina Gorna, Nicolas Iooss, Yannick Seurin, and Rida Khatoun. 2025. Exposing Go's Hidden Bugs: A Novel Concolic Framework. doi:10.48550/arXiv.2505.20183 arXiv:2505.20183 [cs].
- [15] Jia Li, Jiacheng Shen, Yuxin Su, and Michael R. Lyu. 2025. ColorGo: Directed Concolic Execution. doi:10.48550/arXiv.2505.21130 arXiv:2505.21130 [cs].
- [16] Nico Naus, Freek Verbeek, Dale Walker, and Binoy Ravindran. 2023. A Formal Semantics for P-Code. In *Verified Software. Theories, Tools and Experiments*, Akash Lal and Stefano Tetta (Eds.). Springer International Publishing, Cham, 111–128. doi:10.1007/978-3-031-25803-9_7
- [17] NSA. 2017. Ghidra. <https://ghidra-sre.org/>
- [18] OCamlPro. 2025. OCamlPro/owi: Seamless bug-finding for Wasm, C, C++, Rust and Zig. <https://github.com/OCamlPro/owi/tree/main>
- [19] PLSSec. 2024. Haybale: a general-purpose symbolic execution engine written in Rust. <https://github.com/PLSSec/haybale> original-date: 2019-06-18T00:00:21Z.
- [20] R-Fuzz. 2024. SymSan: Time and Space Efficient Concolic Execution via Dynamic Data-Flow Analysis. <https://github.com/R-Fuzz/symsan> original-date: 2021-07-01T00:48:53Z.
- [21] Radare2-org. 2024. Radare2: Libre Reversing Framework. <https://github.com/radareorg/radare2> original-date: 2012-07-03T07:42:26Z.
- [22] Koushik Sen and Gul A. Agha. 2006. Concolic Testing of Multithreaded Programs and Its Application to Testing Security Protocols. <https://hdl.handle.net/2142/11148>
- [23] Christopher Shao, Grace Yin, and Justin Restivo. 2018. DUCKEE GO: Dynamic and User-friendly Concolic Execution Engine in GO. <https://css.csail.mit.edu/6.858/2018/projects/cshao-graceyin-jrestivo.pdf>
- [24] Sigma Prime. 2024. Omni Portal - Security Assessment Report - v2.1. <https://github.com/sigp/public-audits/blob/master/reports/omni-network/Sigma%20Prime%20-%20Omni%20Network%20-%20Omni%20Portal%20-%20Security%20Assessment%20Report%20-%20v2.1.pdf>
- [25] TinyGo-Org. 2025. Documentation TinyGo. <https://tinygo.org/docs/>
- [26] Trail of Bits. 2019. Security assessment techniques for Go projects. <https://blog.trailofbits.com/2019/11/07/attacking-go-vr-tips/>
- [27] Trail of Bits. 2024. MAAT: Dynamic Symbolic Execution and Binary Analysis framework. <https://github.com/trailofbits/maat> original-date: 2021-10-19T09:23:10Z.
- [28] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Providence RI USA, 865–878. doi:10.1145/3297858.3304069
- [29] Fish Wang and Yan Shoshitaishvili. 2017. Angr - The Next Generation of Binary Analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 8–9. doi:10.1109/SecDev.2017.14
- [30] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R. Lyu. 2017. Concolic Execution on Small-Size Binaries: Challenges and Empirical Study. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE/IFIP, 181–188. doi:10.1109/DSN.2017.11 ISSN: 2158-3927.