


Automated Vulnerability Detection in Go: Concolic Execution for Multi-Threaded Binaries

Karolina Gorna[1,2] Nicolas Iooss[2] Yannick Seurin[2]
Rida Khatoun[1] Keith Makan[3]

[1]Telecom Paris, [2]Ledger Donjon, [3]The University of the Western Cape

Tuesday, June 30, 2026

Pass the SALT 2026

Go Language

Go [1] is a high-level programming language,
statically typed and compiled.

Figure: The Go Gopher, mascot of the Go programming language.

Go Usage

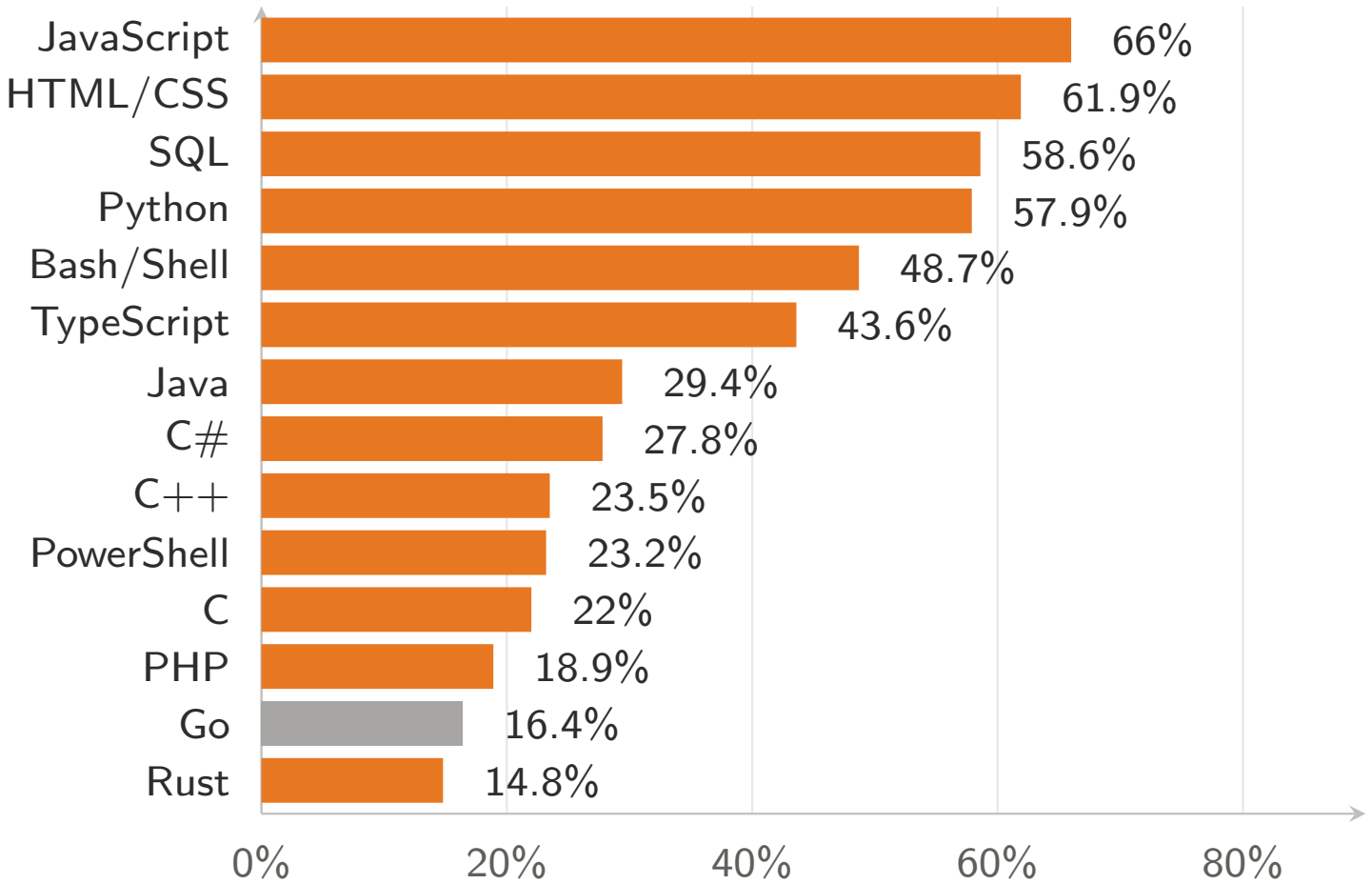

Figure: Most used programming languages among developers. Source: Stack Overflow Developer Survey 2025 ($n = 31{,}771$). Go ranks 13th with 16,4% of users.

Go binaries

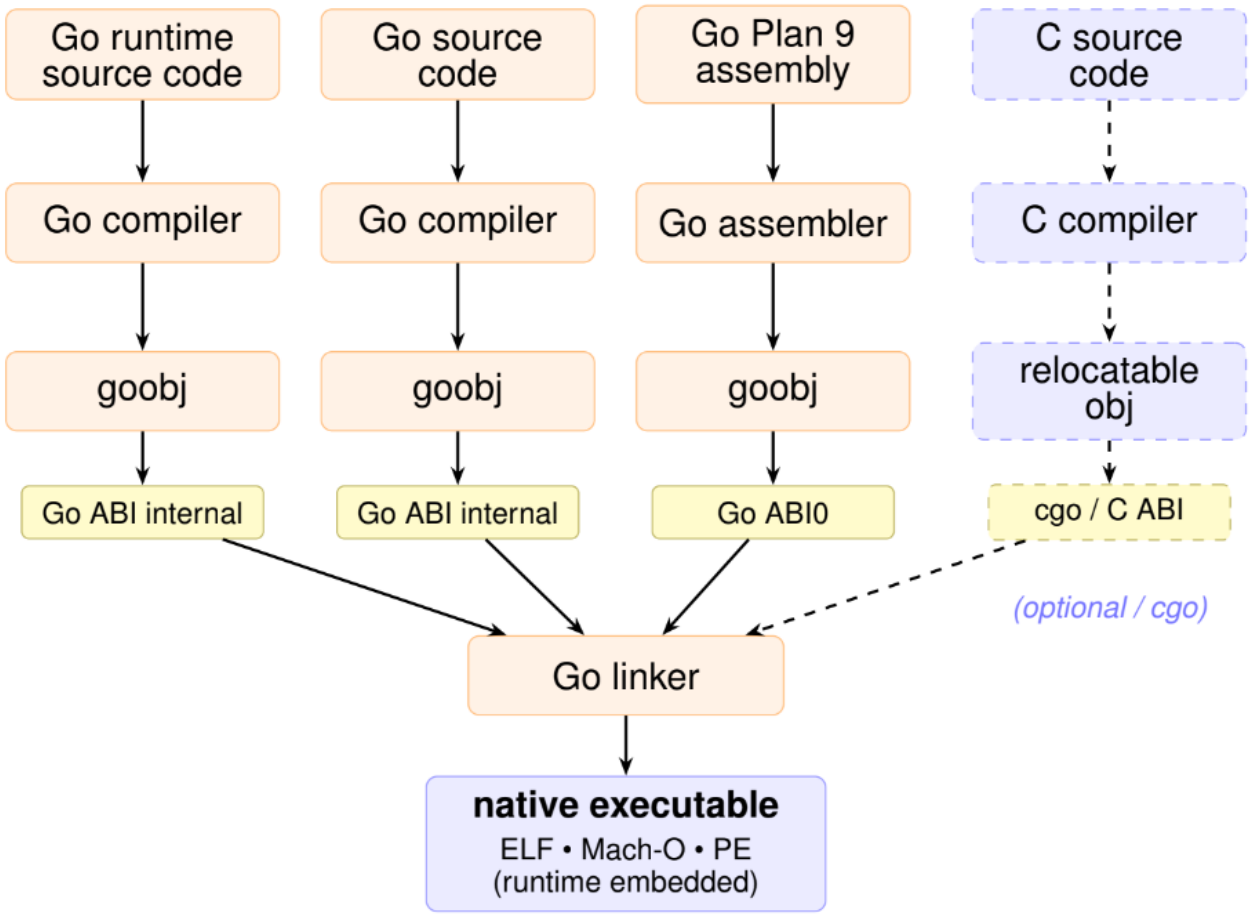

Figure: The Go compiler (*gc*) pipeline

Note. `cmd/compile` produces `goobj` files under ABIInternal (register-based, since Go 1.17); `cmd/asm` uses the legacy stack-based ABI0; the two interoperate via linker-generated wrappers. C code (dashed) is optional via cgo. `cmd/link` produces a statically linked native executable (ELF / Mach-O / PE) with the runtime embedded; this thesis focuses on ELF (Linux/x86-64).

Go Security Tool Landscape

Category	Examples	Strengths / Limitations
Static Analysis	`gosec` [2], `go vet` [3], `staticcheck` [4], `errcheck` [5]	Fast, CI-friendly; misses deep logic bugs
Dependency Scanners	`Snyk` [6]	Finds known CVEs; ignores custom logic flaws
Query-Based (Static)	`CodeQL` [7]	Customizable and powerful; setup/query effort
Fuzzing (Dynamic)	`go test -fuzz`, `GoLibAFL` [8]	Good for crashes; necessity to have a good corpus and harnesses, and do not give a full path coverage
Property-Based Fuzzing	`gopter` [9]	Stateful checks; black-box, limited insight
Specialized Tools	`krf` [10], `on-edge` [11], `nilaway` [12]	Targets `defer/panic`; narrow scope

Go Security Tool Landscape

Category	Examples	Strengths / Limitations
Static Analysis	`gosec` [2], `go vet` [3], `staticcheck` [4], `errcheck` [5]	Fast, CI-friendly; misses deep logic bugs
Dependency Scanners	`Snyk` [6]	Finds known CVEs; ignores custom logic flaws
Query-Based (Static)	`CodeQL` [7]	Customizable and powerful; setup/query effort
Fuzzing (Dynamic)	`go test -fuzz`, `GoLibAFL` [8]	Good for crashes; necessity to have a good corpus and harnesses, and do not give a full path coverage
Property-Based Fuzzing	`gopter` [9]	Stateful checks; black-box, limited insight
Specialized Tools	`krf` [10], `on-edge` [11], `nilaway` [12]	Targets `defer/panic`; narrow scope

Note: Aren't there any **symbolic execution tools** in the Go toolchain ?

Concolic execution

General steps of a software security analysis:

Static Analysis → *Fuzzing* → *Symbolic Execution* → *Formal Verification*

More details about symbolic-based techniques:

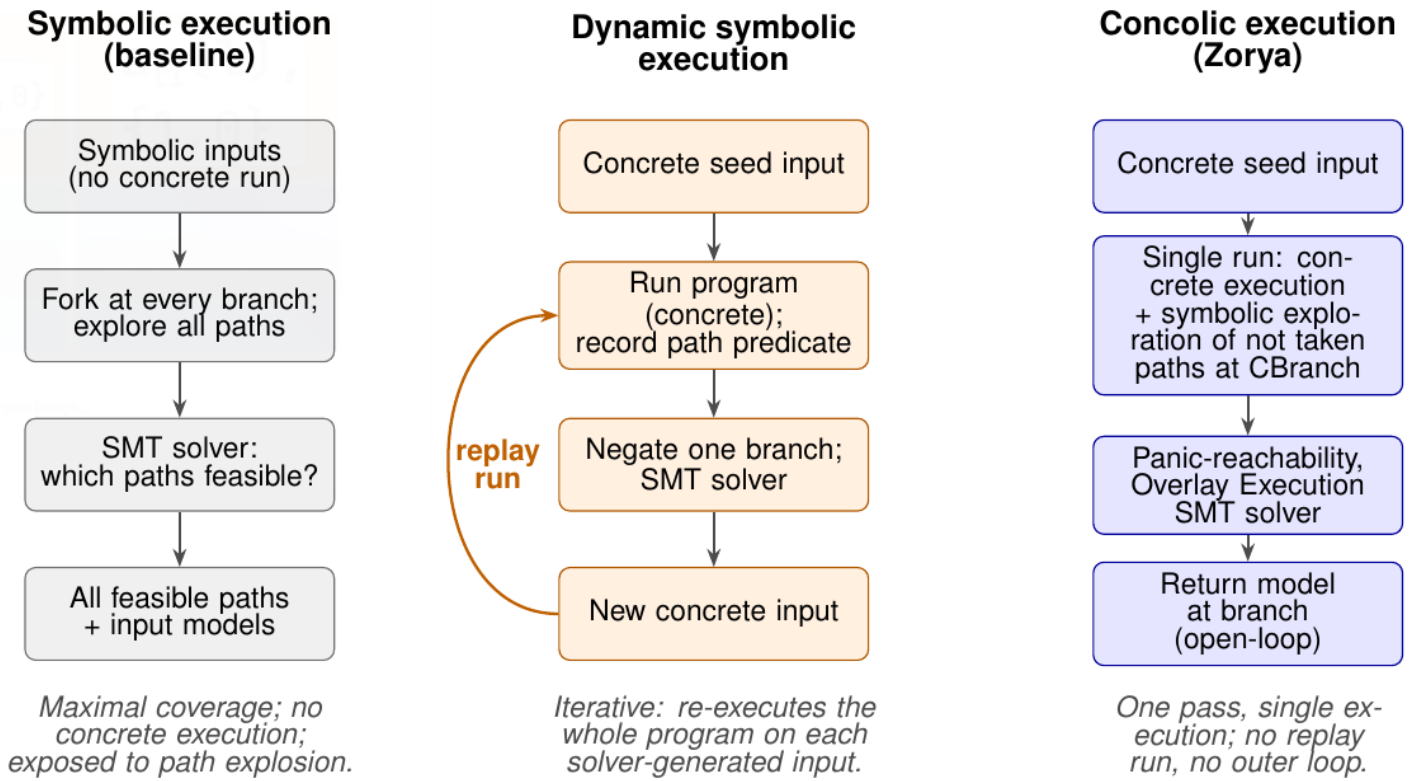

Figure: Overview of the symbolic-based exploration techniques

Comparison of symbolic-based execution engines

Tool	Lang.	Method	Target	IR	Go	Solver	Outer loop
angr	Python	SE or CE	Bin.	VEX, P-Code	No[1]	Z3, Boolector, CVC4	Optional
BINSEC	OCaml	DSE+BB-DSE	Bin.	DBA	No	Z3, Boolector, CVC4	Optional
ConcoLLMic	Py+LLM	Agentic CE	Src+bin	LLM agent	Yes	Z3 (LLM)	Agent-driven
Cottontail	Py+LLM	LLM-driven CE	Src	ESCT	Yes	Z3 (LLM)	LLM-driven
CUTE	C/Java	Iterative CE	Src	Src instr.	No	lp_solve	Yes
DuckEEGO	Go	SE or CE	Src	Go AST	Yes	Z3	Yes
Fuzzolic	C	SE+fuzz	Bin.	QEMU	No	Fuzzy-SAT	Fuzzer
Haybale	Rust	SE	Bin.	LLVM IR	No[2]	Boolector	Optional
KLEE	C++	SE forking	LLVM	LLVM IR	No[2]	STP, Z3, MetaSMT	No
MAAT	C++	SE or CE	Bin.	LLVM IR	No[2]	Z3	Optional
Manticore	Python	Single-path CE	Src+bin	Custom	No	Z3	Optional
MIASM	Python	SE or CE	Bin.	Custom	Yes	Z3, CVC4	Optional
Radius2	Rust	SE+taint	Bin.	ESIL	Yes	Z3, Boolector	Optional
S2E	C++	Selective SE	System bin.	LLVM IR	No	Z3, STP	No
SAGE	C#	Iterative CE	x86 bin	Custom	No	Z3	Yes
SymCC	C++	Compile-time CE	LLVM IR	LLVM IR	No[2]	Z3	Fuzzer
SymQEMU	C++	Compile-time CE	Bin.	TCG	Arch-agn.	Z3	Fuzzer
SymSan	C++	SE or CE	Bin.	LLVM IR	No[2]	Z3	Fuzzer
Triton	C++	DSE	Bin.[3]	AST (own IR)	No[4]	Z3	Optional
Zorya	**Rust**	**CE**	**Bin.**	**P-Code**	**Yes**	**Z3**	**No**

SE = symbolic execution, CE = concolic execution, DSE = dynamic symbolic execution, BB-DSE = backward-bounded DSE, LLM = large language model agent, TCG = QEMU tiny code generator. "SE or CE" = tool supports both modes independently. "Outer loop?" = whether the tool automatically re-runs the binary with solver-generated inputs.
[1]VEX gaps. [2]gollvm unmaintained. [3]x86, x86-64, ARM32, AArch64, RISC-V. [4]No Go runtime models.

What does P-Code look like ?

```
0043d6a0 ba 06 00        MOV         EDX,0x6
         00 00
                                          RDX = COPY 6:8
0043d6a5 f0 0f b1 11     CMPXCHG.... dword ptr [RCX]=>local_2c,EDX
                                            CALLOTHER "LOCK"
                                            $U5480:4 = LOAD ram(RCX)
                                            $Uca900:4 = COPY $U5480:4
                                            CF = INT_LESS EAX, $Uca900:4
                                            OF = INT_SBORROW EAX, $Uca900:4
                                            $Ucaa00:4 = INT_SUB EAX, $Uca900:4
                                            SF = INT_SLESS $Ucaa00:4, 0:4
                                            ZF = INT_EQUAL $Ucaa00:4, 0:4
                                            $U13480:4 = INT_AND $Ucaa00:4, 0xff:4
                                            $U13500:1 = POPCOUNT $U13480:4
                                            $U13580:1 = INT_AND $U13500:1, 1:1
                                            PF = INT_EQUAL $U13580:1, 0:1
                                            CBRANCH <0>, ZF
                                            EAX = COPY $Uca900:4
                                            RAX = INT_ZEXT EAX
                                            BRANCH <1>
                                          <0>
                                            $U5480:4 = COPY EDX
                                            STORE ram(RCX), $U5480:4
                                          <1>
                                            CALLOTHER "UNLOCK"
```

Figure: Exemple of low P-Code code from Ghidra

* CMPXCHG (INTEL instruction) - Compares the value in the AL, AX, EAX, or RAX register with the first operand. If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, EAX or RAX register. RAX register is available only in 64-bit mode.

Zorya, pcode-generator, pcode-parser

- Supports concrete and symbolic data types, x86-64 instructions and system calls
- Allows detection of logic bugs and vulnerabilities in Go binaries
- Generate full execution traces
- Implemented in Rust

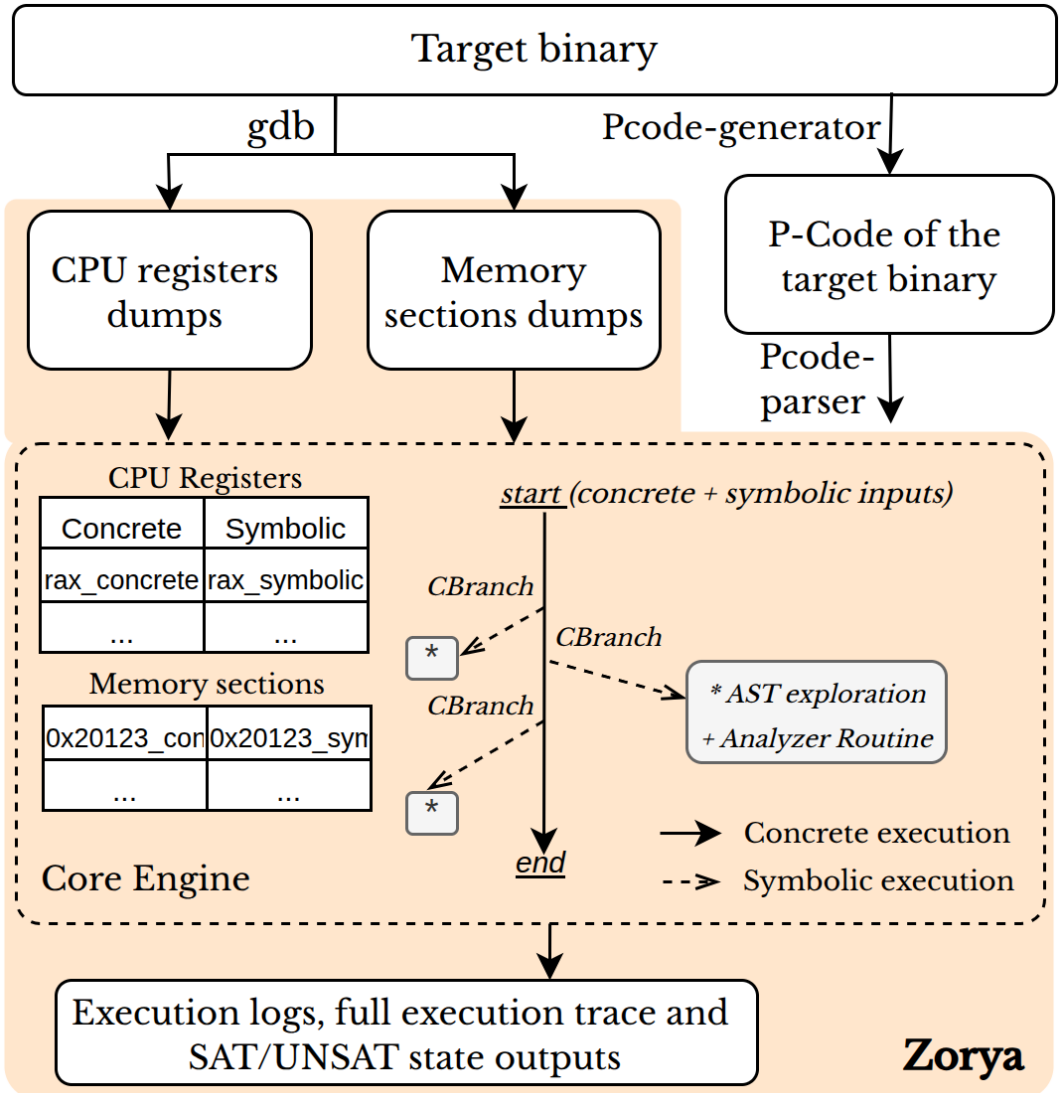

Figure: Overview of the workflow [13]

Analyzer Routine

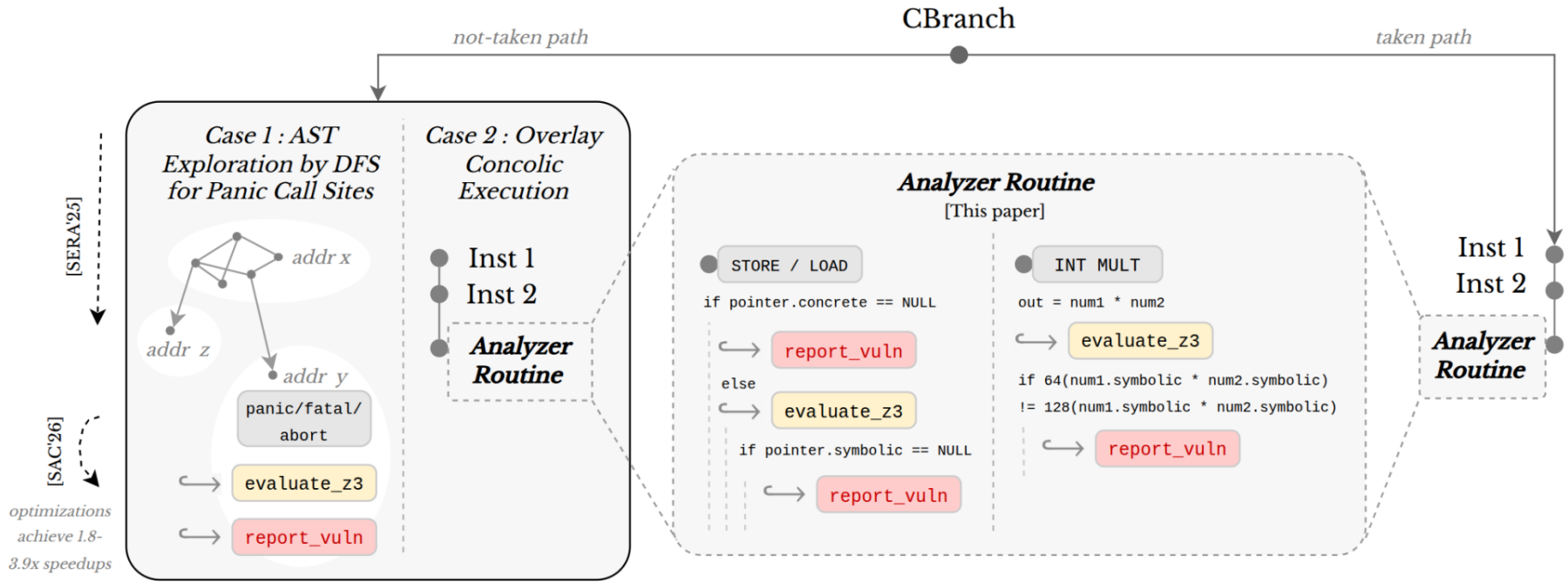

Figure: Overview of Zorya workflow, including AST exploration and Overlay Concolic Execution.

Evaluation

How does Zorya compare to other Go tools and symbolic execution tools regarding bugs detection?

		Size (MB)	staticcheck	gosec	nilaway	go-fuzz	GoLibAFL	Zorya		BINSEC	SymQEMU	KLEE
Technique used			Static Analysis	Static Analysis	Static Analysis	Fuzzing	Fuzzing	Concolic Execution		Symbolic Execution	Symbolic Execution	Symbolic Execution
Needs Additional Files or Automated Execution (Auto.)?			Auto.	Auto.	Auto.	**Harness**	**Harness**	Auto.		**Init Files**	Auto.	**LLVM bitcode**
Outputs the execution trace of each instruction?			No	No	No	No	No	**Yes**		**Yes**	No	**Yes**
Nil Pointer Dereference	kubectl-2025	106	.	.	.	✓	✓	✓	Analyzer routine (LOAD check)	.	.	.
	kubelet-2025	121	.	.	.	✓	✓	✓	Analyzer routine (LOAD check)	.	.	.
	geth-graphql-2025	71	.	.	✓	✓	✓	✓	Panic-reach. (`nilPanic`)	.	.	.
	geth-tracers-2024	66	.	.	✓	✓	✓	✓	Analyzer routine (LOAD check)	.	.	.
Integer Overflow	p224-elliptic-2021	3.3	.	.	.	.	.	.	.	.	.	.
	fasthttp-2020	6.5	.	.	.	.	.	.	.	.	.	.
	tendermint-2018	34	.	.	.	.	.	.	.	.	.	.
	evm-gascost-2017	19	.	.	.	.	.	✓	Analyzer routine (INT_MULT check)	.	.	.
Index Out of Bounds	kube-sm-2025	87	.	.	.	✓	✓	.	.	.	.	.
	coredns-2025	110	.	.	.	✓	✓	✓	Panic-reach. (`panicIndex`)	.	.	.
	goprotobuf-2013	3.8	.	.	.	✓	✓	✓	Panic-reach. (`panicIndex`)	.	.	.

Figure: Evaluation of Go toolchain tools, symbolic execution tools and Zorya against a real-world Go bugs dataset [EASE'26]

Creation of the Zorya MCP Server

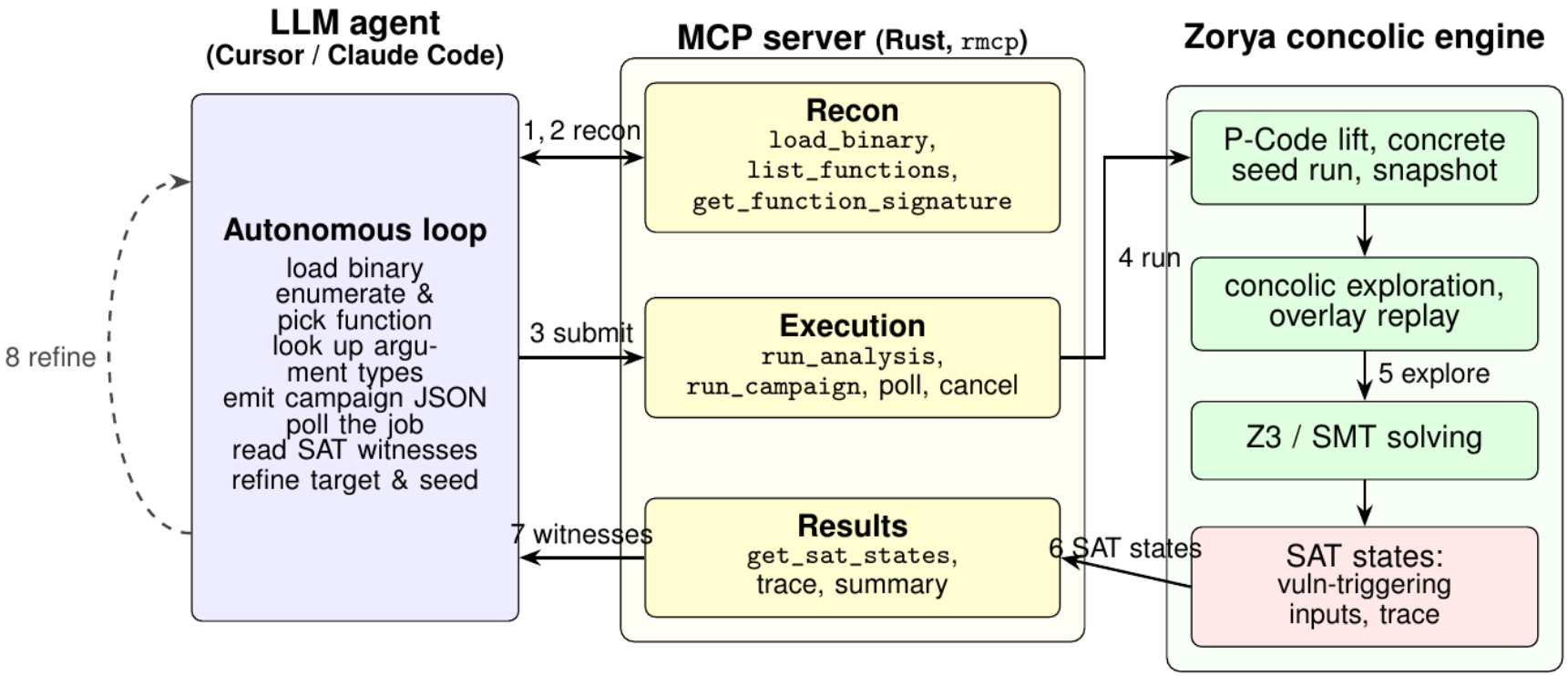

Figure: Overview of the interactions between the Zorya Engine and the Zorya MCP server

Real-world Vulnerabilities Found with Zorya

Project	Bug type	Status	Report
Nil-pointer dereference			
`runatlantis/atlantis`	Nil-ptr deref	Fixed	atlantis #6492
`crossplane/provider-http`	Nil-ptr deref	Open	provider-http #178
`kedacore/keda`	Nil-ptr deref	Fixed	keda #7798
Integer overflow / underflow			
`erpc/erpc`	Integer overflow	Open	erpc #869
`trufnetwork/kwil-db`	Integer overflow	Open	kwil-db #1701
`cometbft/cometbft`	Integer overflow	Fixed	cometbft #5846
`gnolang/gno`	Integer overflow	Open	gno #5639
`XinFinOrg/XDPoSChain`	Integer underflow	Open	XDPoSChain #2362
`kedacore/keda`	Float-to-int overflow	Ongoing	keda #7796
`multiversx/mx-chain-go`	Float-to-int64 overflow	Reported	private report
Index out-of-bounds / slice bounds			
`mandiant/gopacket`	OOB slice	Fixed	gopacket #23
`mandiant/gopacket`	OOB slice	Fixed	gopacket #25
`0xPolygon/bor`	OOB slice	Fixed	bor #2221
`zeromicro/go-zero`	OOB slice	Fixed	go-zero #5607
`zeromicro/go-zero`	Slice bounds	Fixed	go-zero #5614
`seaweedfs/seaweedfs`	Slice bounds	Fixed	seaweedfs #9712
`seaweedfs/seaweedfs`	Index out of range	Fixed	seaweedfs #9713
`multiformats/go-multiaddr`	Slice bounds	Open	go-multiaddr #288
`multiformats/go-multiaddr`	Index out of range	Ongoing	go-multiaddr #289
`nats-io/nats.go`	Slice bounds	Ongoing	nats.go #2100
Security advisory			
`pion/dtls`	ECDHE_PSK panic	Published	CVE-2026-54908
`pion/stun`	XOR-MAPPED-ADDR panic	Published	CVE-2026-54909
`LeJamon/go-xrpl`	MPT `int64` wrap	Published	GHSA-j5cw-qr86-mmv7
`LeJamon/go-xrpl`	MPT scaling overflow	Ongoing	GHSA pending
`gochain/gochain`	Disclosed advisory	Published	GHSA-rmvq-87f6-4pfc

Table: 28 findings across 148 targets (binary + function entry point), May–June 2026 with the Zorya MCP Server. Average cost: 3 € per target identification.

Handling concurrency bugs

- State-of-the-art: focused on scheduling algorithm, code instrumentation, equivalent threads interleavings forms (Mazurkiewicz classes)

Handling concurrency bugs

- State-of-the-art: focused on scheduling algorithm, code instrumentation, equivalent threads interleavings forms (Mazurkiewicz classes)
- Why there is no tool that uses an SMT solver to find concurrency bugs ?

Handling concurrency bugs

- State-of-the-art: focused on scheduling algorithm, code instrumentation, equivalent threads interleavings forms (Mazurkiewicz classes)
- Why there is no tool that uses an SMT solver to find concurrency bugs ?
- Common idea: **the inputs that lead to bug/vuln in a program and the concurrency bugs are orthogonal problems**. Partially false (see the work on POR-SE).

Concurrency-bug Checkpoints & Scheduling

Checkpoint	Bugs exposed	Why it matters	Preferred scheduling
`futex` syscall	Deadlocks; order violations on shared state	A goroutine is about to block or wake, so a misordered wait/signal can stall progress or corrupt shared state	*Fair (FIFO)* for lost-wakeup & starvation; *targeted preemption* between two lock acquisitions to force a cyclic wait
`clone` syscall	Initialisation races; parent–child data races	Parent/child interleaving determines whether initialisation happens before use	*Child-first*: run new goroutine before parent finishes, exposing use-before-initialisation
Memory interactions (atomic ops)	Data races; atomicity violations	Other goroutines may observe an inconsistent intermediate state	*Preemption at a memory barrier*, between a check and its dependent update, to force an unserialisable interleaving

Table: Concurrency-bug checkpoints targeted by Zorya: for each, the bug class exposed, why the checkpoint is critical, and the scheduling strategy that best provokes it.

Volos Race Detector

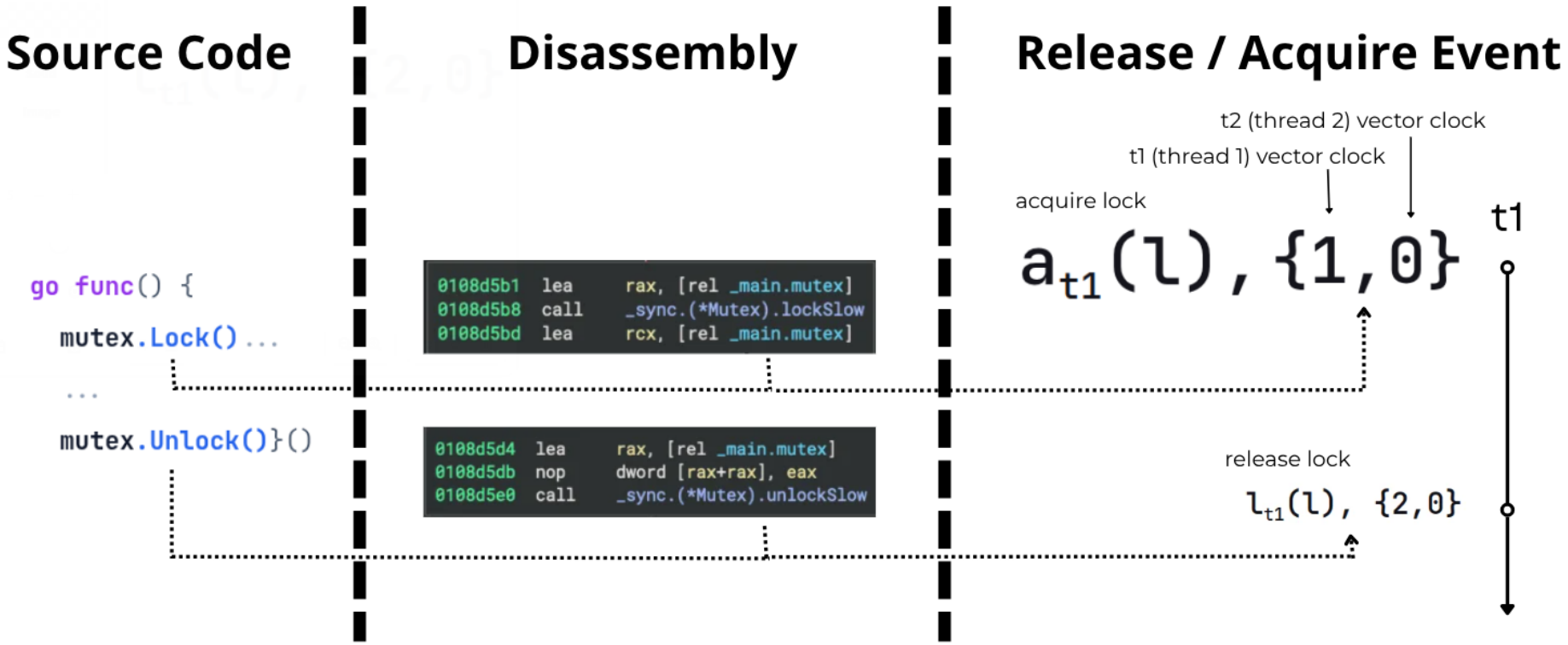

Figure: Overview of the vector clock in Volos, designed to find race conditions

Volos Race Detector

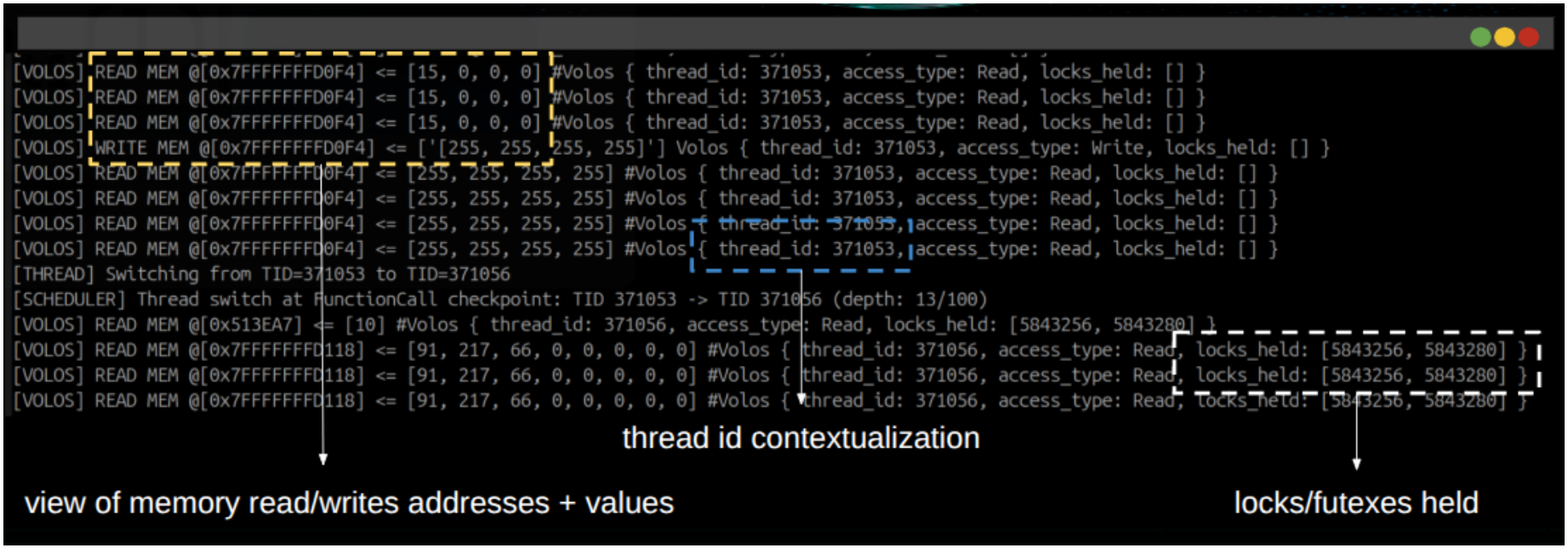

Figure: Overview of Volos execution

Where we currently are?

```go
func dispatch(data []byte) {     // data: symbolic input from the binary
    ch := make(chan int)

    go func() {                  // producer
        for _, b := range data {
            ch <- int(b)         // send -- panics if ch is already closed
        }
    }()
    go func() {                  // control
        if len(data) > 0 && data[0] == 0xCA {
            close(ch)            // early close on a crafted first byte
        }
    }()

    <-ch                         // consume one result
}
```

Listing: An input-triggered send on a closed channel. The control goroutine closes `ch` only on a crafted first byte; if that close is ordered before a producer send, Go panics. The crash needs the right input *and* the right interleaving.

What is next?

- Evaluate the link between inputs and concurrency bugs on C and Go binaries with a real-world dataset (In progress),
- Add functions summaries to accelerate the exploration (In progress),
- Include more types of concurrency bugs, including weak memory bugs? (In progress)
- Keith will be presenting this summer at BSides Johannesburg.

The Plugin Layer

You can implement your own detecting plugin !

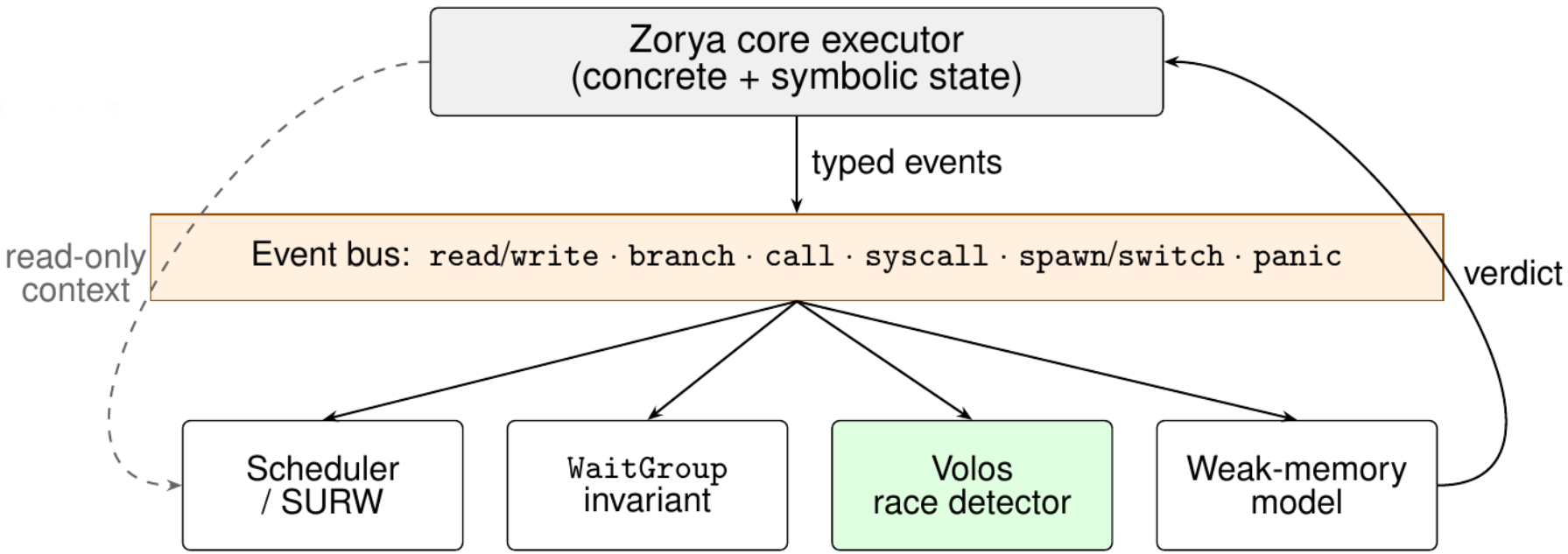

Figure: Overview of the Zorya Plugin Layer

Conclusion

Zorya is the first concolic execution engine using Ghidra's P-Code as IR and optimized for analysing Go binaries.

Evaluated on 11 real-world gc-compiled vulnerabilities from production Go projects and used to find 28 new bugs/vuln in open sourced projects.

Open-sourced contributions:

- **Zorya:** github.com/Ledger-Donjon/zorya
- Evaluation: github.com/Ledger-Donjon/zorya-evaluation
- Dataset: github.com/Ledger-Donjon/logic_bombs_go

Zorya repository

Bibliography I

[1] Google.
The Go Programming Language, 2025.

[2] Securego.
GoSec : Golang static analyzer, 2024.
original-date: 2016-07-18T18:01:08Z.

[3] Google.
GoVet : Golang static analyzer, 2024.

[4] Google.
StaticCheck : Golang static analyzer, 2018.

[5] Kamil Kisiel.
kisielk/errcheck: checking for unchecked errors in Go code., 2024.
original-date: 2013-02-24T22:32:02Z.

[6] Snyk.
snyk/cli, 2024.
original-date: 2015-10-30T11:36:00Z.

[7] CodeQL.
CodeQL, 2021.

[8] Security-Research-Labs.
srlabs/golibafl, February 2026.
original-date: 2025-04-01T15:13:33Z.

[9] Leanovate.
leanovate/gopter: the GOlang Property TestER, 2024.
original-date: 2016-02-11T07:20:49Z.

Bibliography II

[10] Trail of Bits.
trailofbits/krf: a Kernelspace Randomized Faulter., 2024.
original-date: 2018-12-16T20:28:15Z.

[11] Trail of Bits.
trailofbits/on-edge: a library for detecting certain improper uses of the Defer, Panic, and Recover pattern in Go programs., 2023.
original-date: 2019-03-15T10:12:00Z.

[12] Sonal Mahajan.
NilAway: Practical Nil Panic Detection for Go, November 2023.

[13] Karolina Gorna, Nicolas Iooss, Yannick Seurin, and Rida Khatoun.
Exposing Go's Hidden Bugs: A Novel Concolic Framework.
In *2025 IEEE/ACIS 23rd International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 1–6, May 2025.
ISSN: 2770-8209.

Thank you. Questions?

karolina.gorna@telecom-paris.fr

Concolic execution

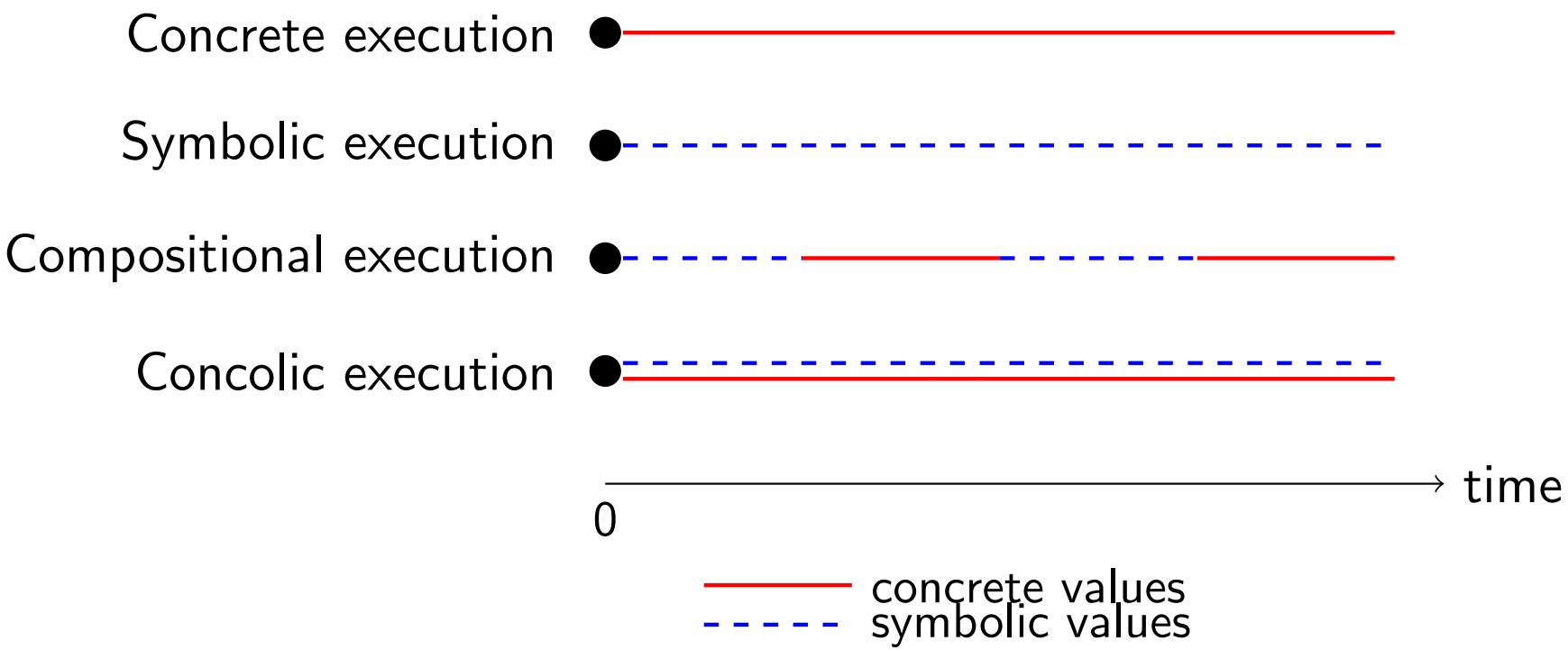

Figure: Different types of executions

Evaluation

Binary	Side bug	Opcode	Depth
(a) Caught during overlay execution			
kubelet-2025	Concrete nil deref	LOAD	3
geth-graphql-2025	Concrete nil write	STORE	2
(b) Depth limit reached, confirmed on same path			
kubelet-2025	Reachable panic	CBRANCH	15
geth-tracers-2024	Reachable panicIndex	CBRANCH	15
kube-sm-2025	Reachable panic	CBRANCH	15
goprotobuf-2013	OOB slice access	LOAD	15
tendermint-2018	Nil pointer deref	LOAD	15
tendermint-2018	Reachable panic (×3)	CBRANCH	15
(c) No side bug: kubectl-2025, coredns-2025, evm-gascost-2017, fasthttp-2020, p224-elliptic-2021			

Figure: Overlay activity across the 11 binaries. (a) side bug caught during overlay execution at the indicated depth; (b) depth 15 reached, then an AST scan + Z3 confirmed a panic site on the same path; (c) no side bug.