



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

Karolina Gorna^{1,2}

¹Télécom Paris (LTCI), ²Ledger Donjon

PhD defense, Thursday, October 8th 2026, 2:00 pm, Télécom Paris

Reviewers David Monniaux (CNRS, Verimag), Jean-Yves Marion (Université de Lorraine, LORIA)

Examiners Lesly-Ann Daniel (EURECOM), Pierre Wilke (CentraleSupélec)

Supervisors Rida Khatoun (Télécom Paris), Yannick Seurin (Ledger Donjon)

Guests Nicolas Iooss (Ledger Donjon), Robin David (Epsilon)

- I. Context: Go Binaries and Concolic Execution
- II. Zorya: Building the Methodology
- III. Zorya: The Evaluation
- IV. Towards Concurrency Bugs Detection
- V. Conclusion

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



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

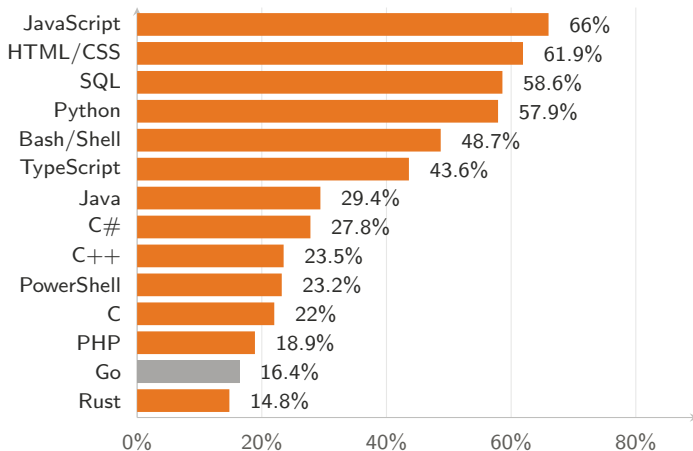


Figure: Most used programming languages among developers. Source: Stack Overflow Developer Survey 2025 ($n = 31,771$). Go ranks 13th with 16,4% of users.

CVEs (i.e. security vulnerabilities) in Go projects, 2020–2026

Category	2020	2021	2022	2023	2024	2025	2026*	Total
Cloud	76	75	131	120	137	118	182	839
Web Infrastructure	25	37	33	21	27	48	214	405
Blockchain	8	12	12	7	12	4	6	61
Go standard library	9	17	34	30	14	32	41	177
Other	82	65	220	290	343	452	1465	2917

Main projects. *Cloud:* Argo CD, Podman, Grafana, Vault, Rancher. *Web infrastructure:* Traefik, Gitea, Fiber, Caddy, oauth2-proxy. *Blockchain:* Evmos, go-ethereum, Hyperledger Fabric, tss-lib.

Source: OSV Go export (Go vulnerability database, GitHub advisories), 7 Oct. 2026. One category per CVE, from the affected module path; CVE-ID year. *2026: partial year.

Go runs critical infrastructure

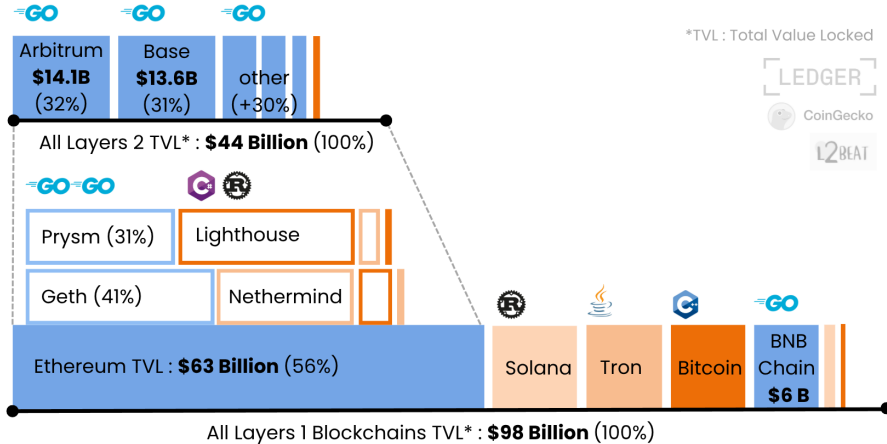


Figure: Go in blockchains: total value locked (TVL) by blockchain and by client. Go logos mark Go software (Arbitrum, Base, Prysm, Geth, BNB Chain). Sources: CoinGecko, L2BEAT, Rated Network, Ethernodes.

Two Go compilers, two kinds of binaries



DONJON

IP PA011

TinyGo

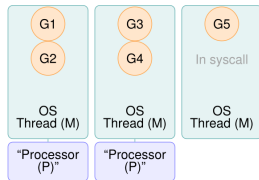
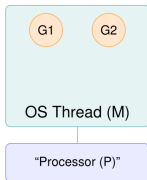
gc

What it is

small compiler for microcontrollers and WebAssembly

the reference compiler, used in production

OS Threads and Goroutines



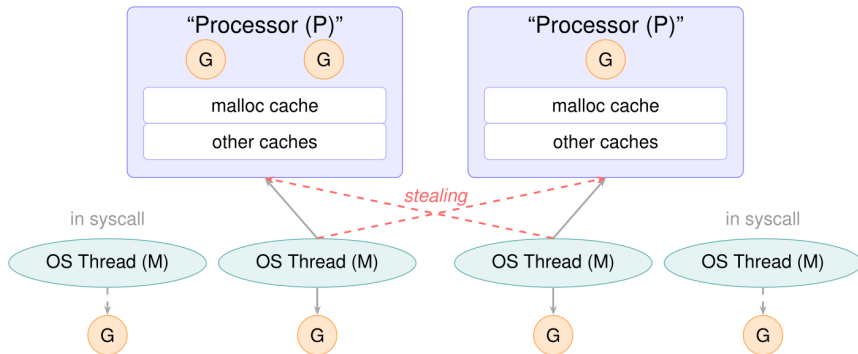
Example bug: an empty pointer (nil)

an explicit check calls a dedicated runtime function (`nilPanic`)

no check: the bug just happens, and the runtime turns the system error into a panic

Other compilers (gccgo, gollvm, llgo, TangoLLVM) are unmaintained, incomplete or experimental.

gc and its G-M-P preemptive scheduler



- A logical processor (**P**) holds a local run queue of goroutines (**G**) and allocator caches.
- An OS thread (**M**) needs a P to run Go code; with an empty queue, it **steals** from another P.
- An M blocked in a system call **releases its P**, so another M can take it.

Source: Nguyen [2,3], figure redrawn.

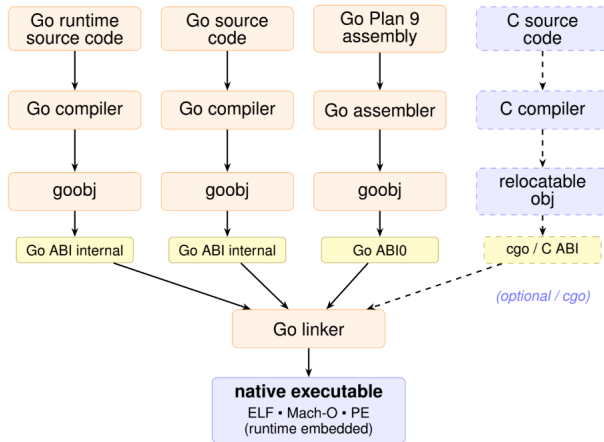


Figure: The Go compiler (*gc*) pipeline

Note. `cmd/compile` produces `goobj` files under `ABInternal` (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).

C binaries vs Go binaries: what makes Go harder

	C binaries	Go binaries
Structure	one standard layout	depends on the compiler (<i>gc</i> or <i>TinyGo</i>)
Runtime	small, the program's own code	a full runtime embedded in the binary
Concurrency	added by a library (threads)	native to the language (goroutines, channels)
Scheduler	the operating system	the Go runtime, preemptive, on top of the operating system

Category	Examples	Limits
Static analysis	gosec [4], go vet [5], staticcheck [6], errcheck [7]	pattern-based , so they miss bugs that depend on specific input values
Query-based	CodeQL [8], Snyk [9]	need the source ; find only what the queries or the known-CVE database describe
Fuzzing	go test -fuzz, GoLibAFL [10], gopter [11]	need a harness and a corpus per target; limited coverage ; random inputs rarely satisfy narrow conditions
Specialised	krf [12], on-edge [13], nilaway [14]	each covers one bug family only (defer, panics, nil flows)

Are there other, more performant methods, to analyze Go binaries ?

Are there other, more performant methods ?

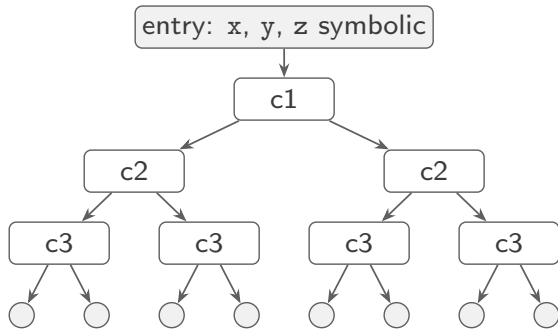
New methods	Limits for Go binary analysis
Chaos engineering, fault injection [15, 16, 17]	needs a live deployment; perturbations are visible on a public network
Model checking	for a quickly changing codebase, a faithful abstraction is hard for the unbounded heap and the Go scheduler
Static symbolic execution	path explosion; heavy solver pressure

Concolic execution

Combines concrete and symbolic execution, and encodes security invariants in the engine itself;
Its limit: one path at a time

Symbolic vs concolic execution

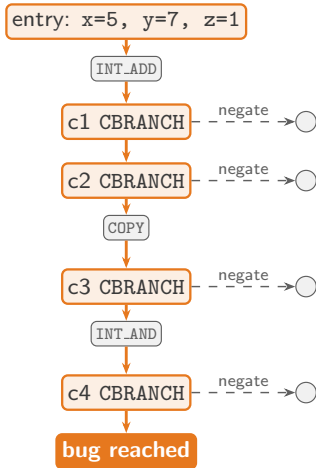
Symbolic execution: explore all paths



2^n paths for n branches:
memory and solver budget exhausted

bug deeper in the binary: never reached

Concolic execution: follow one concrete path,
explore the not taken paths symbolically



inputs are concrete
and symbolic

at each CBRANCH:
explore the other
path, **negate**
the condition, solve,
then return to the
concrete path

- **RQ1.** What must a benchmark contain to evaluate symbolic execution engines on Go binaries?
- **RQ2.** Can concolic execution on binaries find vulnerabilities without source code?
 - **RQ2.1** Which IR is faithful enough for Go?
 - **RQ2.2** Can symbolic arguments tell if an untaken branch can panic?
- **RQ3.** Does it work on real Go binaries?
 - **RQ3.1** Can we analyse multi-threaded binaries?
 - **RQ3.2** Can we detect silent bugs, with no panic?

- I. Context: Go Binaries and Concolic Execution
- II. Zorya: Building the Methodology**
- III. Zorya: The Evaluation
- IV. Towards Concurrency Bugs Detection
- V. Conclusion

Comparison of symbolic-based execution engines

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

¹VEX gaps. ²gollvm unmaintained. ³x86, x86-64, ARM32, AArch64, RISC-V. ⁴No Go runtime models.

What P-Code looks 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
                                CALL OTHER "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>
                                CALL OTHER "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.

- 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

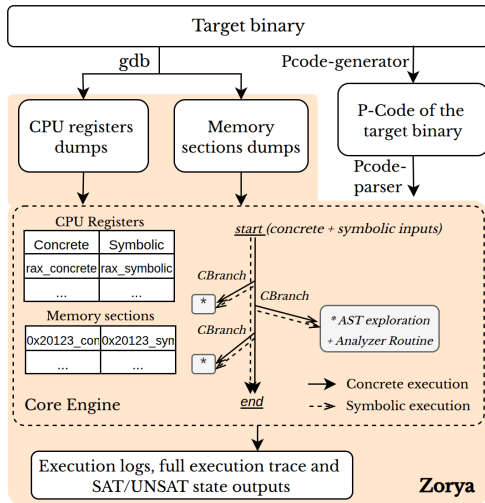


Figure: Overview of the workflow [18]

Overlay concolic execution: explore the untaken branch

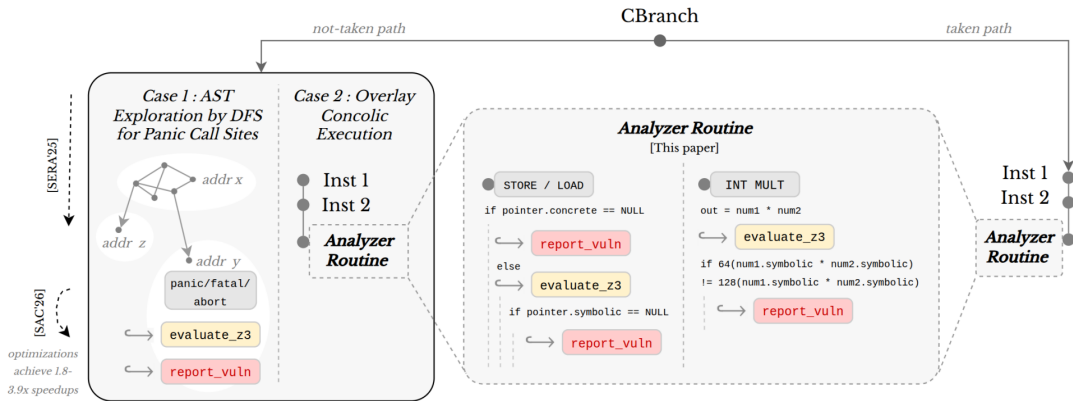


Figure: Zorya workflow with AST exploration and overlay concolic execution. Source: EASE'26

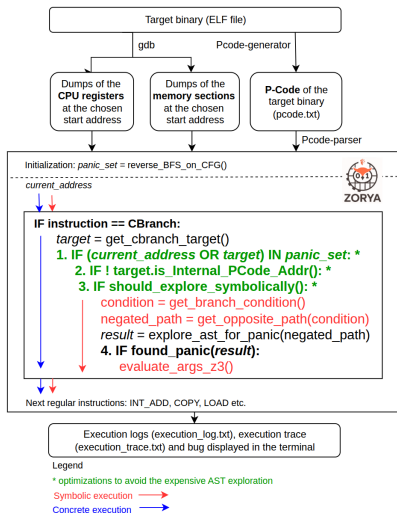


Figure: Panic gating before the Z3 query.

Added optimizations

1. **Panic reachability** (coarse filter): can this branch reach a panic site in the CFG?
2. **Internal targets** (artifact filter): is it a real program branch, not an internal P-Code jump?
3. **Constraint context** (relevance filter): does it depend on a symbolic argument or an earlier constraint?
4. **AST pre-check** (speculative filter): does the negated path hit a panic within X blocks?

Measured on the ACM SAC dataset [19]:

up to two orders of magnitude fewer solver queries;
1.8–3.9× faster when gating discards 33–70 % of branches.

Not always a win for small binaries.

Choice	Problem solved	May miss a bug (FN)	May flag a non-bug (FP)
INPUTS AND MEMORY			
Anchoring concrete slice pointer, $0 \leq \text{len} \leq \text{cap} \leq 64$, strings ≤ 256 B	finite memory, tractable solving	bugs that need longer inputs	arguments no caller would pass (function mode)
Lazy concretisation of symbolic offsets	no symbolic memory	other values of the offset	none: the expression stays in Π
Function mode DWARF arguments	skips runtime init and CLI parsing	stripped binaries; code outside the function	function-isolated findings: hardening, not shown reachable

Design choices and errors (2/2): execution and exploration

Choice	Problem solved	May miss a bug (FN)	May flag a non-bug (FP)
EXECUTION			
One execution no re-run with solver-made inputs	cost grows with one path, not with the number of paths	bugs on paths the seed input does not follow	none
EXPLORATION			
Panic gating + X-block AST look-ahead	solver only where a panic is reachable	panic site farther than X blocks	none: positives are replayed
Overlay replay $N = Y$ blocks	bounded cost per branch	bugs beyond Y blocks	none: positives are replayed

- I. Context: Go Binaries and Concolic Execution
- II. Zorya: Building the Methodology
- III. Zorya: The Evaluation**
- IV. Towards Concurrency Bugs Detection
- V. Conclusion

What is a logic bomb?

A **logic bomb** [20] is a small program whose fault fires **only on a precise input**: the *trigger*.

```
1 num1, num2 := readInputs()
2 if num1 == 5 && num2 == 5 { // trigger
3     var p *int
4     *p = 1                // the bomb
5 }
```

Random inputs rarely reach it: the engine must *solve* the trigger.

	Regular benchmark	Logic bomb benchmark
Label	a bug class	the trigger and the expected outcome
The tool must	flag the weakness	produce an input that triggers it
Checked by	matching the label	replaying the input

Why a new benchmark?

Corpus	Go source	Go binaries	Non-conc. bugs	Logic bombs	Why it does not fit
Xu et al. [20, 21] ("C Logic Bombs")	✗	✗	✓	✓	C calling convention and memory model; no Go runtime
LAVA-M [22]	✗	✗	✓	✓	x86 C only; one injected-bug pattern
Juliet [23]	✗	✗	✓	✗	synthetic, source-level, made for static tools
OSS-Fuzz [24]	✓	partial	✓	✗	crashes without trigger or CWE labels
Tu et al. [25] (Go concurrency)	✓	✗	✗	✗	source-level; concurrency bugs only
LogicBombs-Go	✓	✓	✓	✓	built for this evaluation

25 programs, ten vulnerability types

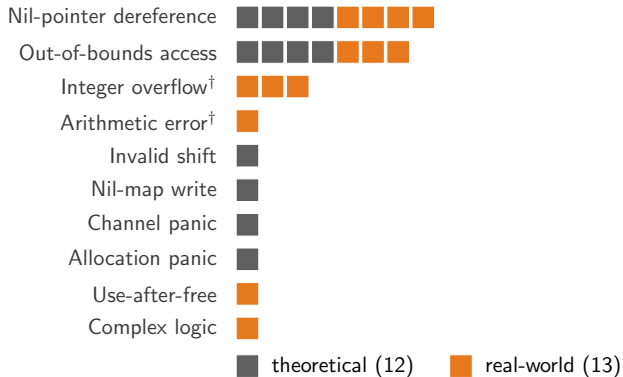


Figure: Programs per vulnerability type, one square per program. [†] silent: wrong result, no panic.

Two compilers. 12 theoretical bombs (2 *gc*, 10 TinyGo); 13 real-world (12 *gc*, 1 TinyGo).

Real-world cases come from Kubernetes, go-ethereum, CoreDNS, fasthttp, Tendermint, golang/protobuf and the Go standard library (P-224, CVE-2021-3114).

Two activation modes. *Direct*: crashes on every run (sanity check). *Conditional*: a specific input must be found.

22 of 25 evaluated. Reserved: *gc* crashme, *gc* invalid-shift, escape-analysis-dangling.

Not covered: concurrency bugs.

Evaluation: 11 real-world gc bugs, 8 other tools

		Size (MB)	staticcheck	gosec	nilaway	go-fuzz	GoLibAFL	Zorya	BINSEC	SymQEMU	KLEE
			Static Analysis	Static Analysis	Static Analysis	Fuzzing	Fuzzing	Concolic Execution	Symbolic Execution	Symbolic Execution	Symbolic Execution
	<i>Technique used</i>		Auto.	Auto.	Auto.	Harness	Harness	Auto.	Init Files	Auto.	LLVM bitcode
	<i>Needs Additional Files or Automated Execution (Auto.)?</i>		No	No	No	No	No	Yes	Yes	No	Yes
	<i>Outputs the execution trace of each instruction?</i>		No	No	No	No	No	Yes	Yes	No	Yes
Nil Pointer Dereference	kubect1-2025	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)	.	.	.

11 real-world Go bugs: Zorya finds **7** with no harness; the other symbolic executors find none.

Source: Gorna et al., EASE 2026.

Average time detection

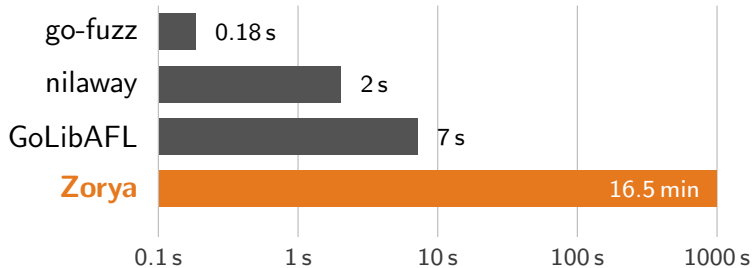


Figure: Average time to detection (log scale).

MCP campaign: who decides what

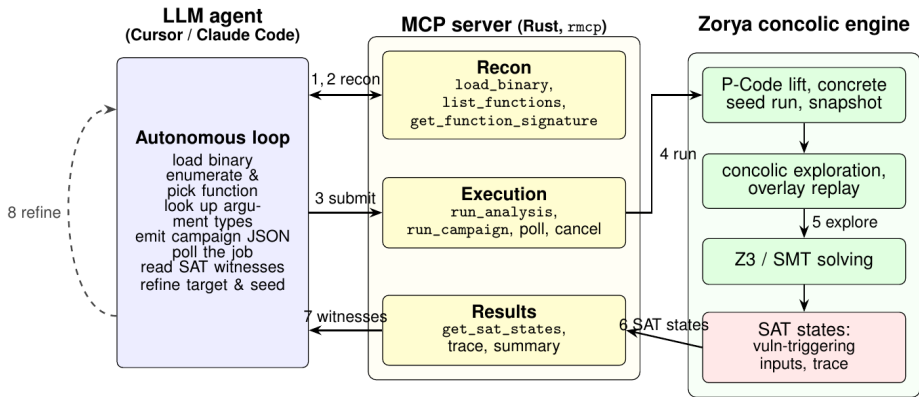
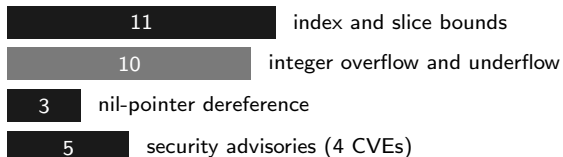


Figure: The Zorya MCP server, a Rust server built with rmcp.

24 findings in production Go projects

By class



By status



Figure: The 24 findings, by class and by status.

Published advisories

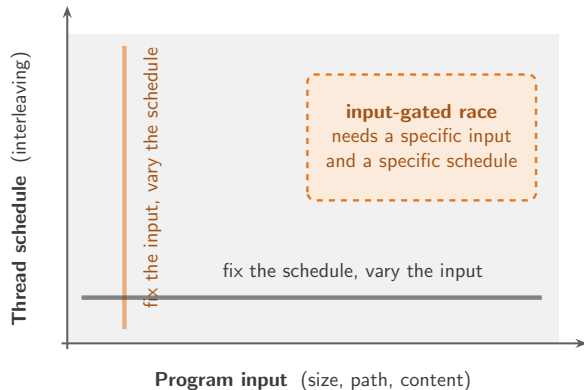
pion/dtls CVE-2026-54908 (medium)
pion/stun CVE-2026-54909 (medium)
go-xrpl CVE-2026-61693 (high),
CVE-2026-61694 (low)
gochain GHSA-rmvq-87f6-4pfc

Campaign of June 2026, Claude Sonnet 4.6; the agent only sees the SAT witnesses. About \$3 per finding, all spent on the agent.

No analysis done on if the agent could find the bugs/vuln without Zorya. Goal: lower the costs.

- I. Context: Go Binaries and Concolic Execution
- II. Zorya: Building the Methodology
- III. Zorya: The Evaluation
- IV. Towards Concurrency Bugs Detection**
- V. Conclusion

The orthogonality assumption



Assumption: whether a concurrency bug exists does not depend on the input.

So tools explore **one axis at a time**: race detectors and model checkers vary the schedule, fuzzers and symbolic execution vary the input.

We aim to show that **this assumption is not always true**.

The plugin layer: three concurrency detectors

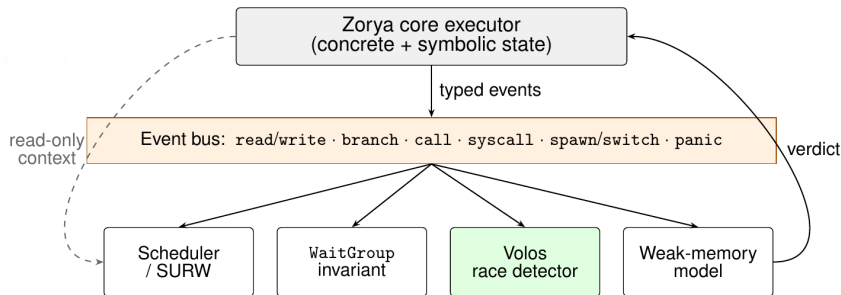


Figure: Plugin layer: detectors subscribe to executor events.

Volos: data races

Two goroutines access the same memory, one of them writes, and no lock protects it. Volos says which input triggers the race.

TOCTOU: check, then use

A file is checked, then used, and can change in between (e.g. stat then open). It also finds checks reached only by a specific input.

chancheck: closed channel

A goroutine sends on a channel that is already closed, which makes Go panic. Z3 solves the input that closes the channel, then sends.

Example: a crafted input races a channel close

```
1 func dispatch(data []byte) { // data: symbolic
2     ch := make(chan int)
3
4     go func() {              // producer
5         v := 0
6         if len(data) > 0 {
7             v = int(data[0])
8         }
9         ch <- v              // panics if ch is closed
10    }()
11    go func() {              // control
12        if len(data) > 0 && data[0] == 0xCA {
13            close(ch)        // early close
14        }
15    }()
16
17    <-ch
18 }
```

A send on a closed channel: a Go-specific synchronisation fault, not a data race.

The panic needs two conditions at once

- the **input**: `data[0] == 0xCA`, so that the control goroutine closes `ch`
- the **schedule**: the close runs before a producer send

Fuzzer	must guess 0xCA and the close-before-send order
Go race detector	sees only the schedule that ran
Pure symbolic execution	finds 0xCA, but not that the close can precede the send

Function summaries: reaching the channel operations

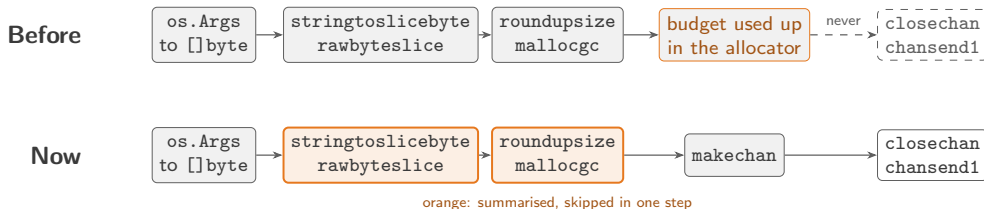


Figure: Reaching the channel operations of `dispatch`, before and after function summaries.

How a summary works, e.g. for `runtime.mallocgc(size, ...)`:

1. At each `call`, look up the callee in the summary table; if absent, run it normally.
2. Read the arguments from the registers (Go ABI: RAX, RBX, RCX).
3. Apply the effect: allocate a zeroed region of the requested size, write its pointer to RAX.
4. Simulate the return: pop the return address and resume after the call, skipping thousands of instructions.

chancheck: the input that makes the send panic

Command (seed input `\x01\x02`: the close is on a branch this run did not take)

```
zorya tests/programs/chancheck-test5-input-close-race/chancheck-test5-input-close-race \  
--lang go --compiler gc --mode main --thread-scheduling all-threads \  
--arg '$'\x01\x02' --negate-path-exploration --plugin chancheck
```

Zorya output

```
[chancheck::send-on-closed-channel] Send on closed channel at 0x49c632 [cross-goroutine]: hchan=0x700000000010, close at 0x49c577 (  
  close on a branch not taken by this run) (pc=0x49c632, severity=Critical)  
Close: tid=3, goid=1002, pc=0x49c577, | $\varphi_{close}$ |=1  
Send: tid=2, goid=1001, pc=0x49c632, | $\varphi_{send}$ |=0  
Triggering input (satisfies  $\varphi_{close} \wedge \varphi_{send}$ ): arg1_byte_0!257 -> #xca  
Path condition  $\varphi = \varphi_{close} \wedge \varphi_{send}$ : (= (ite (= (ite (= arg1_byte_0!257 #xca) #x01 #x00) #x00) #x01 #x00) #x00)  
Interleaving: close (tid=3) ordered before send (tid=2); this run did not schedule it, but nothing orders the two goroutines
```

What chancheck does

1. Record each channel close, with the condition on the input under which it happens (φ_{close}).
2. Record each send, with its own condition (φ_{send}).
3. Ask Z3: is there an input for which both happen ($\varphi_{close} \wedge \varphi_{send}$)? If so, report it with that input.
4. Same goroutine: only if the send follows the close. Two goroutines: if nothing orders them.

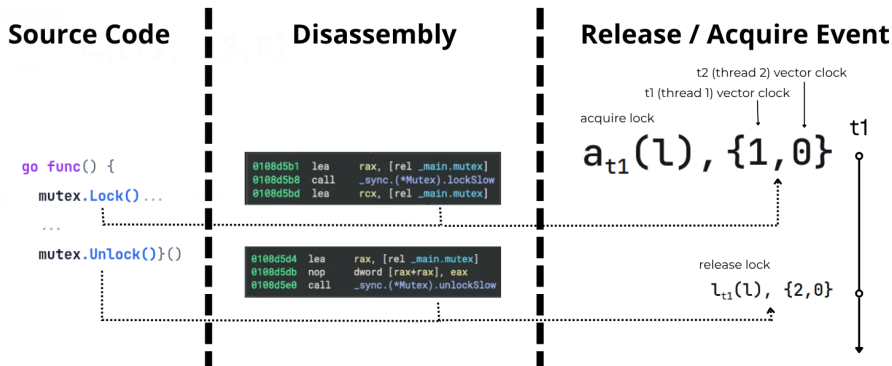


Figure: Overview of the vector clock in Volos, designed to find race conditions. Initial design done with Keith Makan (The University of the Western Cape)

VECT 2.0: “ransomware by design, wiper by accident”

VECT: ransomware-as-a-service launched in Dec 2025; **VECT 2.0** (Feb 2026) has Windows, Linux and ESXi lockers; **25 victims** on its leak site by mid-April 2026.

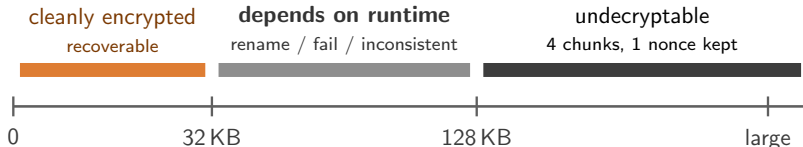


Figure: Outcome of the VECT 2.0 encryption by file size: cleanly encrypted, depending on the runtime, or undecryptable.

The same binary is ransomware for small files and a *wiper* for large ones: the file's **size** decides its fate.

32–128 KB: no chunks. One `ReadFile` copies the whole file (e.g. 64 KB) into the 32 KB buffer. If the bytes after it are mapped and writable (likely), the read **silently overflows**.

Above 128 KB: four chunks, each with its own nonce, but only the last is saved: three quarters are lost.

Source: Check Point [26], LevelBlue [27], JUMPSEC [28], Morphisec [29].

The overflow in memory: a race in VECT's window

In Go, the overflow lands in memory, as in C++

```
1 // read buffer, then path buffer, in one block
2 var state [32 + 128]byte
3 func encryptInPlace(sizeKB int) {
4     if sizeKB <= 128 {           // VECT's only size test
5         for i := 0; i < sizeKB; i++ { // copy the whole file,
6             state[i] = 0x41         // no bound against 32
7         }
8     } else {                     // four chunks, each capped
9         ...
10    }
11 }
12 func worker(sizeKB int, done chan<- bool) {
13     _ = openFile()              // reads the path buffer
14     encryptInPlace(sizeKB)
15     done <- true
16 }
```

Zorya + Volos output (trimmed, seed "033")

```
[volos::input-gated-data-race-unprotected]
Data race at 0x577860 [input-dependent]
  Write vs Write, go=1001 / go=1002
  phi (simplified): 32 < sizeKB <= 128
  triggering input: "111"   escape: "890"
```

0x577860 is state[32]: the first byte past the buffer.

Races iff $32 < \text{size} \leq 128$: VECT's window.

Our check of φ on all 1 000 inputs.

With a plain [32]byte buffer, Zorya instead reports Go's bounds-check panic (input 089): Go stops the overflow.

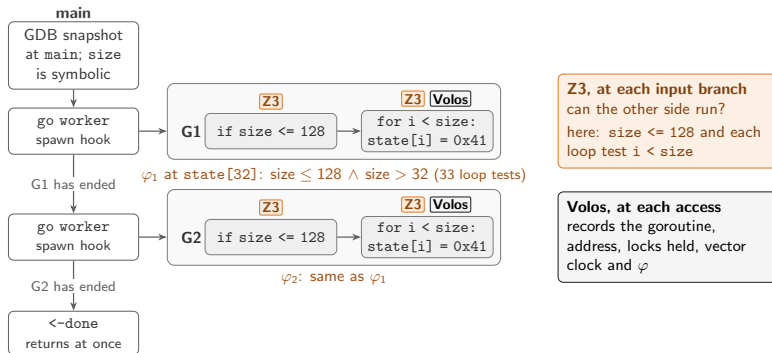
One finding with **both** the input window and the race; the code never compares size with 32.

Assumed: the path buffer follows the read buffer (VECT's layout is not public).

Source: our runs, Zorya 0.0.5, seed 033 (panic: seed 016), all-threads scheduling, Volos.

How Zorya + Volos find a race (1/2): during the run

Window model, seed 033: two goroutines stand for VECT's OS threads. Time flows downwards.



Zorya runs the goroutines **sequentially in a fixed order**, not truly simultaneously. Volos then detects races by recording each access's goroutine ID, address, locks, vector clock and φ .
Not yet: pausing a goroutine midway, other orders, blocking receives.

How Zorya + Volos find a race (2/2): at the end of the run



2. Volos: the race pair

same address, state[32];
no common lock

ordering edges:
main $\rightarrow$ G1, main $\rightarrow$ G2 (spawn)
G1 $\rightarrow$ main, G2 $\rightarrow$ main (done)

none between G1 and G2:
the two writes are unordered

3. Z3: two queries per pair

Can both writes run?
 $\varphi_1 \wedge \varphi_2$ SAT
yes: 111

Can the race be avoided?
 $\neg(\varphi_1 \wedge \varphi_2)$ SAT
yes: 890

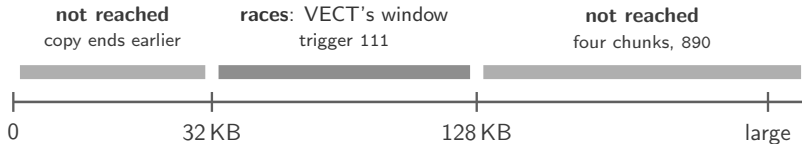
4. Report

Input-dependent race

trigger: 111
escape: 890

this write races iff
 $32 < \text{size} \leq 128$

The write to state[32], by file size

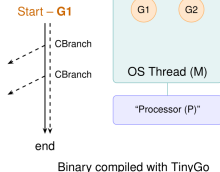


The window comes from the copy loop meeting the 32 KB buffer, and the 128 KB test.

Inside the buffer, byte i races iff $i < \text{size} \leq 128$.

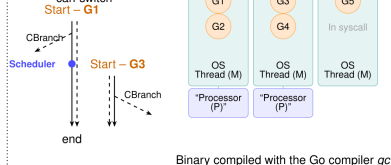
Implementation status: goroutines in TinyGo vs gc binaries

Zorya analysis: on the main OS Thread, on the main goroutine



---> Symbolic analysis
--> Concrete execution

Zorya analysis: on the main OS Thread, on the main goroutine, then the scheduler can switch



Binary compiled with the Go compiler *gc*

What works

TinyGo: one OS thread, several goroutines

main + runtime goroutines; branches explored on the main one

gc: several OS threads and goroutines

switch from G1 to G3; the new goroutine runs to completion at `go f()`

gc: branch exploration inside G3

works if the condition depends on tracked inputs; shown end-to-end on the VECT and DECAF hand-off models

Status

Done

Done

Done

Not implemented yet

Scheduler switching partway through G1

Exploring alternative execution orders

Processor (P) layer, goroutines parked in run queues, syscall handoff

Today: cooperative, not preemptive. Switches only at `go f()`, and back to the parent when the goroutine exits.

Concurrency-bug checkpoints & scheduling

Checkpoint	Bugs exposed	Why it matters	Preferred scheduling
futex syscall	Deadlocks; order violations	A goroutine is about to block or wake; a misordered wait/signal stalls progress or corrupts shared state	<i>Fair (FIFO)</i> for lost-wakeup; <i>targeted preemption</i> between two lock acquisitions
clone syscall	Initialisation / parent/child races	Parent/child interleaving decides whether init happens before use	<i>Child-first</i> : run the new goroutine before the parent finishes
Memory / atomic ops	Data races; atomicity violations	Other goroutines may observe an inconsistent intermediate state	<i>Preemption at a memory barrier</i> , between a check and its dependent update

For each checkpoint: the bug class exposed, why it is critical, and the schedule that best provokes it.

Next step: switch between scheduling algorithms

For bug finding, Zorya will host several **randomized schedulers**, selectable per run.

What a scheduler needs

(A) the next sync event of each goroutine

to build: recognise sync points in P-Code

(B) how many events each goroutine runs

counted in the seed run

Scheduler	(A)	(B)	Idea
POS [30]	✓		random priorities on racing events
URW [31]		✓	random walk weighted by remaining events
SURW [31]	✓	✓	uniform over a chosen subset of events
PCT [32]	✓		depth-bounded baseline
DFS			exhaustive, as an oracle

(A)/(B) decomposition refined with Dylan Wolff (NUS). Reference code: SURW artifact [31].

- I. Context: Go Binaries and Concolic Execution
- II. Zorya: Building the Methodology
- III. Zorya: The Evaluation
- IV. Towards Concurrency Bugs Detection
- V. Conclusion**

- **RQ1** (Which benchmark?) We built LOGICBOMBS-GO: 25 programs with a known trigger and outcome, ten vulnerability types, two compilers, checked in CI.
- **RQ2** (Concolic on binaries?) It finds Go vulnerabilities without source code.
 - **RQ2.1** (Which IR?) P-Code is faithful enough for Go.
 - **RQ2.2** (Link between symbolic arguments and bugs/vuln?) We check whether an untaken branch can reach a panic, and report the values that lead to it with Z3 SMT solver.
- **RQ3** (Real Go binaries?) It works: 7 of 11 real-world bugs on *gc*, and 24 findings through the MCP server (16 fixed, 5 advisories, 4 CVEs).
 - **RQ3.1** (Multi-threading?) Multi-threaded *gc* binaries can be analysed, thanks to thread state recovery, preemption neutralisation and concolic overlay execution.
 - **RQ3.2** (Silent bugs?) Detected: a Go-Ethereum overflow that no other tool finds.

- **Scalability.** Every instruction before the bug is executed concretely and symbolically. The overlay concolic execution has a fixed depth.

- **Scalability.** Every instruction before the bug is executed concretely and symbolically. The overlay concolic execution has a fixed depth.
- **No constraint accumulation across nested overlays.** Zorya does not yet track the conjunction of all negated conditions.

- **Scalability.** Every instruction before the bug is executed concretely and symbolically. The overlay concolic execution has a fixed depth.
- **No constraint accumulation across nested overlays.** Zorya does not yet track the conjunction of all negated conditions.
- **Math libraries and concolic execution.** `math/big` stores numbers in a dynamic `[]uint64` slice (`nat`): the engine must follow multi-word carries, reallocations and variable-length loops, which give huge Z3 expressions.

- **Scalability.** Every instruction before the bug is executed concretely and symbolically. The overlay concolic execution has a fixed depth.
- **No constraint accumulation across nested overlays.** Zorya does not yet track the conjunction of all negated conditions.
- **Math libraries and concolic execution.** `math/big` stores numbers in a dynamic `[]uint64` slice (`nat`): the engine must follow multi-word carries, reallocations and variable-length loops, which give huge Z3 expressions.
- **Input model.** Slices are capped at 64 elements and strings at 256 bytes; argument recovery needs DWARF (no stripped binaries).

- **Scalability.** Every instruction before the bug is executed concretely and symbolically. The overlay concolic execution has a fixed depth.
- **No constraint accumulation across nested overlays.** Zorya does not yet track the conjunction of all negated conditions.
- **Math libraries and concolic execution.** `math/big` stores numbers in a dynamic `[]uint64` slice (`nat`): the engine must follow multi-word carries, reallocations and variable-length loops, which give huge Z3 expressions.
- **Input model.** Slices are capped at 64 elements and strings at 256 bytes; argument recovery needs DWARF (no stripped binaries).
- **Concurrency.** Interleavings are not explored systematically yet; atomics follow sequential consistency, not Go's weaker memory model.

Scale the engine	symbolic summaries for heavy runtime helpers; checkpointing of shared prefixes; symbolisation of local variables; calibrated overlay depth
Drop the debug-info dependency	recover function names and argument layouts from <code>pcIntab</code> (GoReSym, GoResolver) on stripped binaries
Close the concurrency gap	randomized schedulers (SURW, POS) in a scheduler plugin; runtime invariants as SMT queries; validation on Tu et al. and GoBench; Go weak-memory model
Combine with other tools	hybrid fuzzing and concolic execution; bounded verification from the same symbolic machinery
Broaden the target	other Go versions, ARM64, floating point; C and C++ through the same overlay
Apply to malware	real samples in an isolated lab; quantify how often a concurrency bug is tied to an input, with a corpus labelled on both axes

Conference papers

ACM EASE 2026

From TinyGo to gc Compiler: Extending Zorya's Concolic Framework to Real-World Go Binaries

ACM SAC 2026

Zorya: Automated Concolic Execution of Single-Threaded Go Binaries

IEEE/ACIS SERA 2025

Exposing Go's Hidden Bugs: A Novel Concolic Framework

Book chapters

Springer SCI 2026

Concolic Execution Optimized for Go Binaries using Ghidra's P-Code

Annales des Mines 2025

La menace dans le champ des cryptoactifs

Oral presentations

Black Hat Asia 2026

Zorya: Go Binary Vulnerability Detection with Concolic Execution

Pass the SALT 2026

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

DefMal 2026

The Data Behind the Race: Rethinking How We Separate Program Inputs from Concurrency Bugs



[LEDGER]
DONJON



Thank you. Questions?

karolina.gorna@[ledger, telecom-paris].fr

github.com/Ledger-Donjon/zorya

github.com/Ledger-Donjon/zorya-evaluation

github.com/Ledger-Donjon/logic_bombs_go



Zorya repository

Questions and assumptions

anticipated questions; faithfulness assumptions

Context and methods

Go in malware; method comparison;
chaos-engineering tools; dynamic symbolic vs
concolic execution

Benchmark and Zorya results

full LOGICBOMBS-GO table; panic-gating timings;
overlay side findings; a silent bug (memoryGasCost)

Campaign

campaign file; the 24 findings

Concurrency: landscape

positioning; requirements; benchmarks (C/C++,
Go); race detectors on binaries

VECT and DECAF

VECT 2.0 in brief; the race inside the encryptor;
DECAF from launch to wipe; threads and
lightweight concurrency

Models of VECT and DECAF for Zorya

why not run Zorya on VECT; one model in three
languages; DECAF worker pool; why Zorya misses
it; a simplified model

Volos and plugins

Volos in action; TOCTOU

Analysing malware in practice

analyst workflow; recovering function names; C++
limits; input and concurrency bugs; the deliberate
version

Q1. The evaluation uses main-only scheduling. Would round-robin change the results, especially for interleaving-dependent bugs?

Very little for the evaluated bugs: they are single-goroutine (overflows, nil dereferences, out-of-bounds), so round-robin mostly adds cost. For interleaving bugs, Zorya runs *one fixed order* (no preemption, no alternative orders): Volos still flags data races from ordering and locksets, but send-on-closed-channel or deadlocks are found only if that order triggers them.

Q2. Overlay paths can give false positives from under-constrained symbolic variables. An example, and how do you validate a witness?

Typical case: a nil dereference on a pointer or interface argument, made symbolic in function mode with no caller constraint, while by Go convention no caller passes nil. Validation: replay the witness concretely (unit test, or registers set at function entry in GDB), check that the value can come from attacker input at the call sites, then re-run with fewer free variables (whole-program mode or caller preconditions). If it is still satisfiable and the replay succeeds, we treat the path as reachable.

Q3. Why does Zorya find the Go-Ethereum overflow that gosec misses?

Semantic, path-sensitive reasoning instead of syntactic matching. gosec's G115 rule only flags conversions: it reports `uint64(mem.Len())` (`gas_table.go`, line 44) and misses `newMemSizeWords * newMemSizeWords` on the next line. Zorya zero-extends both operands to 128 bits and asks Z3 whether the upper 64 bits can be non-zero under the path constraints, including the `MaxUint64-32` guard: satisfiable, with `newMemSize = 0xe2049180867b10c2`. The overflow is silent, so fuzzers without a `big.Int` oracle are likely to miss it too.

Q4. Fuzzers find the same 7 of 11, and faster. What does Zorya add?

No harness: a (function, seed) pair is enough. It solves deep equality predicates, and its generic check finds the silent memoryGasCost overflow that fuzzers without a big.Int oracle miss. It also emits an instruction-level trace. It is slower (16.5 min on average against 7s at most) and misses kube-sm. In practice the tools chain: static analysis picks starting points, Zorya confirms reachability with a satisfying input, a fuzzer broadens coverage around it.

Q5. Can the language model produce false positives?

No by construction: it only proposes (function, seed) pairs. Every positive is a Z3 model, replayed concretely on the binary. A bad proposal costs coverage, not correctness.

Q6. Are the 24 findings exploitable?

Six have a credible path from untrusted input (four CVEs, one gochain advisory, one go-multiaddr panic). The other 18 come from exercising a function in isolation and are hardening opportunities; *fixed* means the maintainers agree the function should be robust at its boundary.

Q7. Why was chaos engineering abandoned?

It needs a live deployment, and on a public network the perturbations are observable: an exploitable failure could be reused before the clients are patched. Concolic execution needs no deployment and leaks nothing.

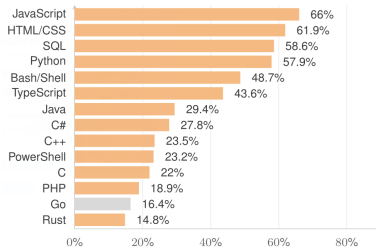
	Assumption	Evidence in the thesis
A1	Sleigh lifting and the P-Code handlers follow the x86-64 semantics	P-Code generated outside Ghidra compared with Ghidra's own on 20 binaries (Go 10/10, C 9/10); detections are replayed
A2	Z3 answers its bit-vector queries correctly	off-the-shelf solver; models are replayed on the binary
A3	DWARF gives the right argument layout in function mode	binaries built with debug information; stripped binaries unsupported

Under A1–A3, a SAT model replayed on the binary is a confirmed finding.
The converse does not hold: absence of a finding is not a proof of absence.

Backup: How much malware is written in Go?

7 %

of all malware samples
CrowdStrike, mid-2021



+2 000 %

new Go malware
2017 to 2020

developers using each language, 2025: Go 13th (16.4 %), ahead of Rust

Ransomware written in Go

Snatch
2018

eCh0raix
2019

Ekans
2020

Hive
2021

DECAF
2021

BlackByte
2022

Figure: Go in the malware landscape: share of samples, growth, developer survey and Go ransomware families.

A **small share** of malware, but growing, with a steady line of Go ransomware.

Source: CrowdStrike [33], Stack Overflow Developer Survey 2025, $n = 31,771$ [34], Intezer [35], Morphisec [36].

Backup: comparison of software-analysis methods for Go binaries



Method	Principle	Advantages	Limits for Go binaries
Static analysis	reason over code or IR without running it	sound over-approximation; full path coverage	needs source or a typed IR; imprecise on interfaces, goroutines, stripped binaries
Fuzzing	mutate inputs, watch for crashes	high throughput; cheap to set up	stalls on deep equality predicates; no control over the scheduler
Software fault injection	inject faults into a running service	exercises error-handling paths	needs a live deployment; leaks attack signals on public networks
Chaos engineering	inject infrastructure disruptions	reveals distributed-system failures	needs a deployed system; ethical and leakage concerns
Dynamic binary instrumentation	rewrite native instructions at run time	no source needed; substrate for other tools	high overhead; not an analysis on its own
Taint analysis	track data flow from sources to sinks	pinpoints attacker-controlled flows	static variant imprecise; dynamic variant unsound and costly
Model checking	explore a finite-state abstraction	sound for the abstracted model	faithful abstraction is hard for the unbounded heap and goroutine scheduling
Static symbolic execution	explore paths purely symbolically	maximal path coverage; synthesises inputs	path explosion; heavy solver pressure
Dynamic symbolic execution	symbolic reasoning along a concrete run	mitigates path explosion; opaque operations via concrete values	unsound on unvisited branches; solver-bound
Concolic execution	one concrete path with a symbolic shadow	scales; synthesises crashing inputs	one path at a time; solver-bound on complex constraints

Backup: chaos-engineering tools considered, and set aside

Tool	Year	Level	What it does (always on a live deployment)
Chaos Monkey [15]	2026	infrastructure	randomly terminates production instances to force resilience
ChaosETH [16]	2023	system call	injects syscall error returns into Ethereum clients (22 error classes, GoEthereum and Nethermind); compares 15 application metrics
Attacknet [17]	2024	network	orchestrates Chaos Mesh faults (clock skew, packet drop, partition, I/O latency) on Kurtosis devnets for Ethereum 2.0 clients

All three perturb a running system. For external vulnerability research they raise a deontological concern,
so the thesis uses them only to delineate its scope.

Backup: Dynamic symbolic execution vs concolic execution

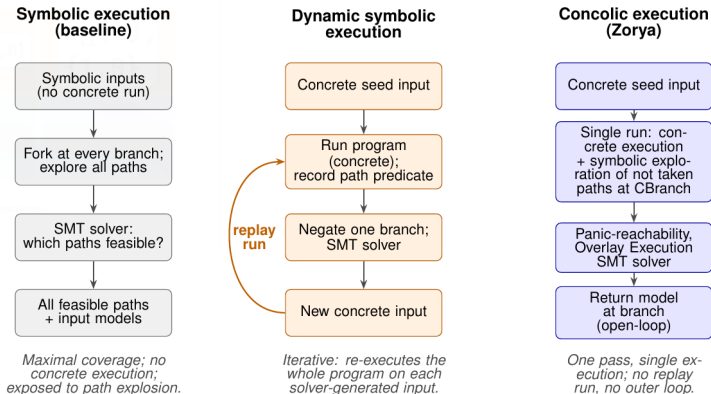


Figure: Overview of the symbolic-based exploration techniques

Dynamic symbolic execution: $prefix \wedge \neg branch$

Concolic execution: $prefix \wedge \neg branch \wedge violation$ (the model is directly a **bug-triggering input**, so no replay is needed)

Backup: the LOGICBOMBS-Go corpus, 25 programs

Theoretical bombs (12)

Type	Binary	Comp.	Mode	Trigger
Nil deref	crashme	gc	C	<code>argv[1][0]=='K'</code>
Nil deref	crashme	TinyGo	C	<code>argv[1][0]=='K'</code>
Nil deref	broken-calculator	TinyGo	C	<code>num1==5 && num2==5</code>
Nil deref	tinygo_additiongo	TinyGo	D	<code>always</code>
OOB	invalid-shift	gc	C	<code>argv[1][0]>=64</code>
OOB	invalid-shift	TinyGo	C	<code>argv[1][0]>=64</code>
OOB	panic-index	TinyGo	C	<code>n<0 n>=3</code>
OOB	tinygo_index-out-of-range	TinyGo	D	<code>always</code>
Shift	tinygo_negative-shift	TinyGo	D	<code>always</code>
Nil map	tinygo_assign-to-nil-map	TinyGo	D	<code>always</code>
Channel	tinygo_too-large-channel	TinyGo	D	<code>always</code>
Alloc	panic-alloc	TinyGo	C	<code>alloc ≥ heap limit</code>

C: conditional; D: direct.

Real-world case studies (13)

Type	Binary	Comp.	Source
Nil deref	kubectl-nil-delegate-2025	gc	Kubernetes kubectl
Nil deref	kubelet-empty-flag-2025	gc	Kubernetes kubelet
Nil deref	geth-graphql-2025	gc	go-ethereum GraphQL
Nil deref	geth-tracers-2024	gc	go-ethereum tracers
Overflow [†]	evm-gascost-2017	gc	go-ethereum EVM gas
Overflow [†]	fasthttp-parseuint-2020	gc	fasthttp Content-Length
Overflow [†]	tendermint-voting-2018	gc	Tendermint voting power
Arith. [†]	p224-elliptic-2021	gc	Go stdlib, CVE-2021-3114
OOB	ksm-oob-2025	gc	kube-state-metrics
OOB	coredns-loop-oob-2025	gc	CoreDNS loop plugin
OOB	goprotobuf-overflow-2013	gc	golang/protobuf
UAF	escape-analysis-dangling	gc	CVE-2020-8945
Logic	omni-vuln4	TinyGo	Merkle-tree sibling OOB

All conditional. [†] silent: needs an oracle. Reserved: gc crashme, gc invalid-shift, escape-analysis-dangling.

Backup: panic-gating speedups (ACM SAC subset)

Binary	Start	Before (s)	After (s)	Gated/Total	Speedup
crashme	main	45.1	35.6	0/4	1.3×
	crash()	23.6	35.8	0/1	0.7×
invalid-shift	main	53.5	36.2	0/5	1.5×
	shift()	55.9	40.6	0/2	1.4×
panic-index	main	489.9	203.1	19/58	2.4×
	index()	148.3	38.3	8/17	3.9×
broken-calculator	main	726.6	331.1	34/97	2.2×
	coreEngine()	77.6	44.1	7/10	1.8×
omni-vuln4	main	>7200	>7200	–	–
	GetMP()	273.1	83.1	12/32	3.3×

Gated/Total: conditional branches discarded by panic-reachability gating over all those encountered. Averages of five runs on inputs that do not trigger a panic.

Binary	Side bug	Opcode	Depth
<i>(a) Caught during overlay execution</i>			
kubelet-2025	Concrete nil deref	LOAD	3
geth-graphql-2025	Concrete nil write	STORE	2
<i>(b) Depth limit reached, confirmed on same path</i>			
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
<i>(c) No side bug: kubectl-2025, coredns-2025, evm-gascost-2017, fasthttp-2020, p224-elliptic-2021</i>			

Figure: Side bugs found by overlay execution: (a) caught during the overlay, (b) depth limit reached, (c) none.

10 side findings in 6 binaries.

- (a) 2 caught during the overlay itself, at depth 2 and 3.
- (b) 8 where the depth limit (15) was reached and an AST scan confirmed a reachable panic.
- (c) 5 binaries with no side finding.

The depth-15 cases suggest several bugs sit just beyond the limit: the depth deserves a principled calibration.

Cache rule across the overlay boundary. SAT answers (a nullable pointer) are kept, UNSAT answers are cleared: the negated branch may relax the constraints that made a pointer safe.

Backup: a silent bug, Go-Ethereum memoryGasCost

```
1      func memoryGasCost(mem *Memory,
2      newMemSize uint64) (uint64, error) {
3          if newMemSize > MaxUint64-32 {
4              return 0, errGasUintOverflow
5          }
6          newMemSizeWords := toWordSize(
7              newMemSize)
8          square := newMemSizeWords *
9              newMemSizeWords
10         linCoef := newMemSizeWords *
11             MemoryGas
12         quadCoef := square / QuadCoeffDiv
13         fee := linCoef + quadCoef - mem.
14             lastGasCost
15         return fee, nil
16     }
```

geth v1.6.0. The guard protects `toWordSize`, not the squaring on line 6: it wraps without any panic, and an attacker allocates huge EVM memory for almost no gas.

What Zorya does.

The squaring is a P-Code `INT_MULT`. Both 64-bit symbolic operands are zero-extended to 128 bits; Z3 is asked whether the upper half can be non-zero under the path constraints, guard included.

Result: SAT in 0.39 s, witness `newMemSize = 0xe2049180867b10c2`.

No oracle written by hand. Fuzzers without a `big.Int` oracle miss it; `gosec` flags the conversion on the line before and not the product.

The counter-example:

`p224-elliptic-2021` (CVE-2021-3114) gives a wrong result without overflow. Zorya misses it.

```
{
  "global": {
    "language": "go",
    "compiler": "gc",
    "binary_path": ".../broken-calculator",
    "thread_scheduling": "main-only",
    "negate_path_flag": true
  },
  "tests": [
    { "id": "main.main", "mode": "main",
      "start_address": "0x4bef60", "args": "2_+3" },
    { "id": "main.coreEngine", "mode": "function",
      "start_address": "0x4bec40", "args": "2_+3" }
  ]
}
```

The unit of analysis is a **(function, concrete seed)** pair, not the whole program.

A file fully determines an analysis: re-running it reproduces the result, and it is the provenance record of each finding.

Written by hand for a batch campaign, or emitted by the MCP server from its recon step.

Backup: the 24 findings

Project	Bug	Status
<i>Nil-pointer dereference</i>		
runatlantis/atlantis	nil deref	Fixed
crossplane/provider-http	nil deref	Fixed
kedacore/keda	nil deref	Fixed
<i>Integer overflow / underflow</i>		
gochain/gochain	overflow (GHSA-rmqv-87f6-4pfc)	Fixed
erpc/erpc	overflow	Ongoing
trufnetwork/kwil-db	overflow	Fixed
cometbft/cometbft	overflow	Fixed
gnolang/gno	overflow	Ongoing
XinFinOrg/XDPoSChain	underflow	Fixed
kedacore/keda	float-to-int overflow	Closed
multiversx/mx-chain-go	float-to-int64 overflow	Reported (private)
LeJamon/go-xrpl	int64 wrap, MPT amount (CVE-2026-61693)	Ongoing
LeJamon/go-xrpl	uint64 mul overflow, MPT amount (CVE-2026-61694)	Fixed

Project	Bug	Status
<i>Index out-of-bounds / slice bounds</i>		
mandiant/gopacket	OOB slice (#23)	Fixed
mandiant/gopacket	OOB slice (#25)	Fixed
0xPolygon/bor	OOB slice	Fixed
seaweedfs/seaweedfs	slice bounds (PR #9712)	Fixed
seaweedfs/seaweedfs	index out of range (PR #9713)	Fixed
zeromicro/go-zero	OOB slice (#5614)	Fixed
zeromicro/go-zero	OOB slice (#5607)	Fixed
multiformats/go-multiaddr	slice bounds (PR #289)	Ongoing
multiformats/go-multiaddr	slice bounds (#288)	Ongoing
pion/dtls	index OOB (CVE-2026-54908)	Fixed
pion/stun	OOB slice (CVE-2026-54909)	Fixed

Status as of October 7th, 2026 (issue and pull-request trackers; Zorya repository, section Findings). Ongoing: fix proposed or in progress, not yet merged. Reported: awaiting the maintainers or the vendor. Closed: closed as not planned.

Backup: Positioning: two axes, one open diagonal

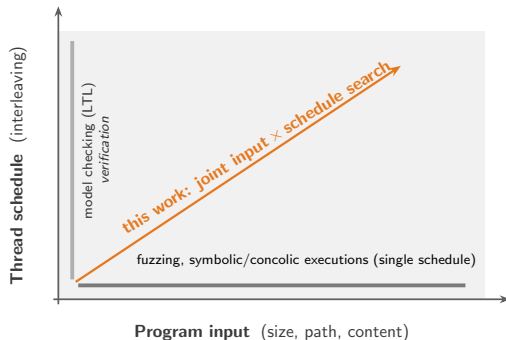


Figure: Verification, fuzzing, symbolic execution and this work, in the input $\times$ schedule plane.

Verification walks up the y-axis. Fuzzing and symbolic execution walk along the x-axis.
This work targets the diagonal: a joint exploration of inputs and schedules.

Backup: What does searching the diagonal require?

Starting point: a **binary**, no source, and a race whose triggering input we want to **compute**.

Requirement	Why
1. Read the binary	malware ships no source
2. Compute the input	a gate like $32 < \text{size} \leq 128$ KB is solved, not guessed
3. Stay tractable	a Go binary embeds its whole runtime
4. See races	the bug needs two goroutines on shared memory

How do existing symbolic engines meet these requirements?

C and C++ corpora

Corpus	Input explored?	Schedule explored?
DataRaceBench [37] 208 OpenMP kernels	✗ the input only sets the array size (one exception, next slide)	✓ race detectors
SCTBench [38] 52 pthread bugs	✗ one fixed input per program	✓ controlled scheduler
Logic bombs [20, 21] 5 of 53 use threads	✓ solve for the triggering input	✗ never labelled

In the corpora we examined, each explores **one axis only**.

```
int main(int argc, char *argv[]) {  
    int N = 100;           // default input  
    if (argc > 1)  
        N = atoi(argv[1]); // the only input  
    int *A = malloc(sizeof(int) * N);  
    #pragma omp parallel for shared(A)  
    for (int i = 0; i < N; i++) {  
        A[i] = i;  
        if (N > 10000)  
            A[0] = 1;      // write-write race  
    }  
}
```

In the code: the input decides.

The race on $A[0]$ exists only if $N > 10\,000$.

In the benchmark: it never fires.

Labelled **race = yes**, but the harness runs
 $N = 32$ to 1024 .

The only input-gated race we found in the 208 kernels
(from OMPRacer, SC'20); none in the 103 Go kernels.

A dynamic detector **cannot report a race whose input it never receives.**

Backup: Benchmarks on concurrency bugs: Go

Corpus	How one test is run	Input explored?	Schedule explored?
Tu et al. [25] 171 bugs, 6 projects	not run: a study of fix patches	✗ not recorded	described, not run
GoBench [39] 103 kernels, 82 real bugs	<code>go test -run TestX</code> <code>TestX(t *testing.T)</code> takes no data	✗ 0 of 103 kernels read args, flags, stdin or files	✓ repeated runs
Uber [40] over 2000 races	<code>go test -race</code> 100K+ existing unit tests	✗ values hard-coded in each test	✓ observed, not controlled
GFuzz [41] 184 new bugs	<code>go test</code> 8199 existing unit tests	✗ test data never mutated	✓ reorders channel messages

In the corpora we examined, no bug is labelled with *both* its triggering input and its interleaving.

Backup: Existing tools: race detectors on binaries

Study	Target	Year	Analysis	RCDS	BFF	Symbolic	Input from a solver
BinTSan [42]	C/C++	2024	LSA/HBA	OTF	Yes	No	✗
Taskgrind [43]	C/C++	2024	HBA	PM	Yes	No	✗
HardRace [44]	C/C++	2024	LSA/HBA	PM	Yes*	Yes	✗
COMRace [45]	C/C++	2022	LSA	CT	Yes	Yes	✗
BiRD [46]	C/C++	2022	LSA/HBA	PM	Yes	No	✗
BinGo [47]	Go	2022	LSA	PM	Yes	No	✗
ConVul [48]	C/C++	2019	HBA	OTF	Yes	Yes	✗
Zorya	Go	2026	LSA/HBA	OTF	Yes	Yes	✓

LSA / HBA = lockset / happens-before analysis. RCDS = race detection style: on-the-fly (OTF), post-mortem (PM) or compile time (CT). BFF = binary format friendly.

Where a detector is marked *Symbolic*, symbolic reasoning serves the race analysis, **not the program input that reaches the race**. As far as we know, only Zorya returns that input.

Source: table by Keith Makan (University of the Western Cape), Black Hat Asia 2026; last column added by us.

Timeline

- Dec 2025** RaaS launched on a Russian-language forum
- Feb 2026** VECT 2.0: Windows, Linux and ESXi lockers
- Apr 2026** Check Point publishes the encryption flaw; Morphisec then documents the thread race

Built with

C++, statically compiled, ChaCha20.
42 encryptor threads on an 8-CPU host.

Impact

25 victims on the leak site by mid-April 2026.

Paying does not help: files over 128 KB cannot be decrypted, even by the operator.

Source: Check Point [26], LevelBlue [27], Morphisec [29].

Backup: The race inside the encryptor

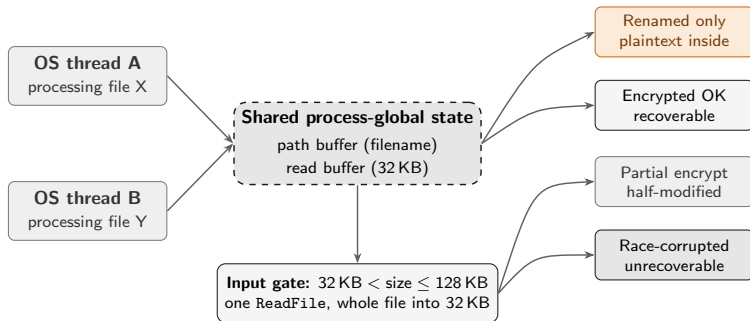


Figure: Two OS threads of VECT share a path buffer and a 32 KB read buffer: the outcome depends on the file size and on the schedule.

Outcome = $f(\text{input size, schedule})$: a thread overwrites shared state while another still uses it.
VECT renames the file to .vect before opening its path: every outcome already has the extension.

Source: Morphisec [29].

Backup: DECAF: a Go ransomware, from launch to wipe

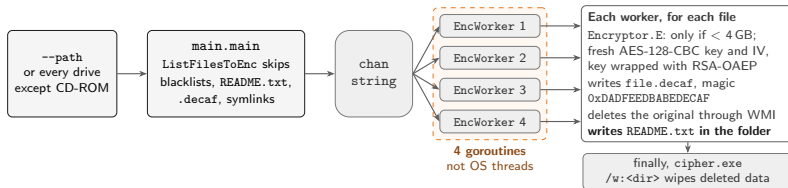


Figure: DECAF, from launch to wipe: one channel feeds four `EncWorker` goroutines.

late Sep 2021	debug build found, symbols and a test assertion kept
days later	stripped pre-release, strings and function names obfuscated, uploaded to test its detection score
within a week	weaponised build on a Morphisec customer site

Go 1.17, the first release that passes arguments in registers: harder to decompile.

Debug build: 98272cad...a7a6f63 (SHA-256).

Four **goroutines**, one `README.txt` per folder: the **red line** hides a race (next slide).

Source: Morphisec Labs, 28 Oct 2021 [36].

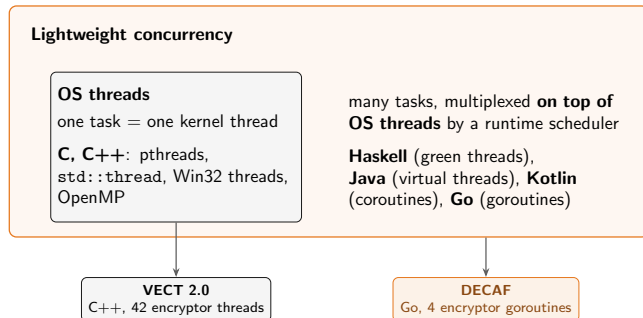


Figure: Concurrency models: one task per OS thread (C, C++; VECT 2.0), or many tasks multiplexed on OS threads by a runtime scheduler (Go; DECAF).

VECT 2.0 races between the **OS threads** it creates.

DECAF races between **goroutines**: the Go runtime multiplexes them onto its own OS threads.

Source: Morphisec [29, 36].

Backup: Why not run Zorya on VECT itself?

It would run the sample

Zorya starts from a GDB
snapshot of the *running* binary

→ isolated lab: future work

No source

VECT's C++ source
was never published

→ public reports only

No C++ support yet

--lang c++ takes the C
path: nothing C++-specific

→ next slide

So we rebuilt **VECT's bug structure** as a small, harmless program,
and wrote it three times: in **C++**, **C** and **Go**.

Source: Zorya repository [19].

Backup: One VECT model, three languages

Each version: 2 workers, one global 32 KB buffer, VECT's 128 KB branch, the file size from argv.

	OS threads, like VECT		goroutines
	C++ (std::thread)	C (pthreads)	Go
Workers run	✗	✓	✓
Shared-buffer race	✗	✓	✓
Its input condition	✗	✗ lost at thread start	✓ solved by Z3
Overflow	✗	✗	✓ 089: VECT's window
Why	library calls return 0: new gives NULL	concrete thread argument; no bounds check	bounds check compiled in

On OS threads, Zorya finds VECT's race but **loses its input condition**.

With goroutines, it finds **both bugs**: that is why the model is in Go.

Source: our runs, Zorya 0.0.5, seed 016, all-threads scheduling, Volos.

Faithful model (gc, -N -1; condensed)

```
1 var readme [2]byte // README.txt of folders 0 and 1
2 func EncWorker(paths <-chan file, done chan<- bool) {
3     for f := range paths {
4         if f.sizeGB >= 4 { // Encryptor.E: < 4 GB only
5             continue
6         }
7         readme[f.dir] = 'R' // writeNote: shared, no lock
8     }
9     done <- true
10 }
11 func main() {
12     // os.Args[1] = "abc": sizes in GB (symbolic) of
13     // files 0 and 1 (folder 0) and file 2 (folder 1)
14     paths := make(chan file)
15     for i := 0; i < 4; i++ { go EncWorker(paths, done) }
16     for _, f := range files { paths <- f }
17     close(paths) // then wait for the 4 workers
18 }
```

Kept: 4 worker goroutines on one channel, the 4 GB filter, one note per folder.

Dropped: file I/O, encryption, deletion.

Ground truth (go -race, 5 runs per input):

Input	Folder 0	Raced
111, 119	two files under 4 GB	1/5, 3/5
391, 931	one file under 4 GB	0/5

The input decides if the race *can* happen; the schedule decides if it *does*.

Backup: Why Zorya cannot find the worker-pool race yet

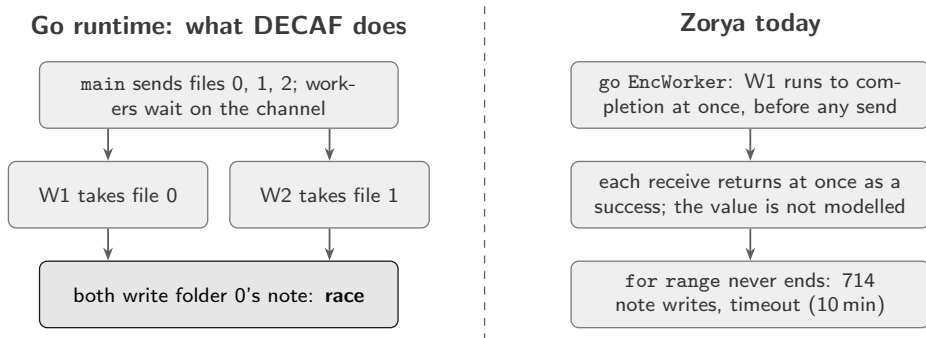


Figure: What DECAF does in the Go runtime (left) and what Zorya does today on the same worker pool (right).

The race needs **two workers to each take a file of the same folder**:
an interleaving that a run-to-completion scheduler cannot produce.

Backup: A simplified DECAF model that Zorya can analyse

Hand-off model: one goroutine per file, no channel

```
1 var readme [2]byte // README.txt of folders 0 and 1
2 func EncWorker(f file, done chan<- bool) {
3     if f.sizeGB < 4 { // Encryptor.E: < 4 GB only
4         readme[f.dir] = 'R' // writeNote: shared, no lock
5     }
6     done <- true
7 }
8 func main() {
9     // os.Args[1] = "abc": sizes in GB (symbolic)
10    for _, f := range files { go EncWorker(f, done) }
11    // then wait for the 3 workers
12 }
```

What changes: which worker takes which file is fixed.
The schedule axis is gone; the shared note and the input gate stay.

Zorya + Volos output (trimmed, 85 s run, seed "111")

```
[volos::input-gated-data-race-unprotected]
Data race at 0x5772a6 [input-dependent]
  Write vs Write, go=1001 / go=1002
  phi (simplified): size0 < 4 /\ size1 < 4
  triggering input: "001"   escape: "801"
1 data race(s) found
```

0x5772a6 is main.readme[0], folder 0's note; go=1001 and 1002 handle files 0 and 1. Third digit: our choice, it is not in φ . No race on folder 1's note: one writer.

Found: the race and its exact input gate, two files under 4 GB in the same folder.

Backup: Volos in action

```
[VOLOS] READ MEM @0x7FFFFFFFD0F4 <= [15, 0, 0, 0] #Volos { thread_id: 371053, access_type: Read, locks_held: [] }
[VOLOS] READ MEM @0x7FFFFFFFD0F4 <= [15, 0, 0, 0] #Volos { thread_id: 371053, access_type: Read, locks_held: [] }
[VOLOS] READ MEM @0x7FFFFFFFD0F4 <= [15, 0, 0, 0] #Volos { thread_id: 371053, access_type: Read, locks_held: [] }
[VOLOS] WRITE MEM @0x7FFFFFFFD0F4 <= ['[255, 255, 255, 255]'] #Volos { thread_id: 371053, access_type: Write, locks_held: [] }
[VOLOS] READ MEM @0x7FFFFFFFD0F4 <= [255, 255, 255, 255] #Volos { thread_id: 371053, access_type: Read, locks_held: [] }
[VOLOS] READ MEM @0x7FFFFFFFD0F4 <= [255, 255, 255, 255] #Volos { thread_id: 371053, access_type: Read, locks_held: [] }
[VOLOS] READ MEM @0x7FFFFFFFD0F4 <= [255, 255, 255, 255] #Volos { thread_id: 371053, access_type: Read, locks_held: [] }
[VOLOS] READ MEM @0x7FFFFFFFD0F4 <= [255, 255, 255, 255] #Volos { thread_id: 371053, access_type: Read, locks_held: [] }
[THREAD] Switching from TID=371053 to TID=371056
[SCHEDULER] Thread switch at FunctionCall checkpoint: TID 371053 -> TID 371056 (depth: 13/100)
[VOLOS] READ MEM @0x513EA7 <= [10] #Volos { thread_id: 371056, access_type: Read, locks_held: [5843256, 5843280] }
[VOLOS] READ MEM @0x7FFFFFFFD118 <= [91, 217, 66, 0, 0, 0, 0, 0] #Volos { thread_id: 371056, access_type: Read, locks_held: [5843256, 5843280] }
[VOLOS] READ MEM @0x7FFFFFFFD118 <= [91, 217, 66, 0, 0, 0, 0, 0] #Volos { thread_id: 371056, access_type: Read, locks_held: [5843256, 5843280] }
[VOLOS] READ MEM @0x7FFFFFFFD118 <= [91, 217, 66, 0, 0, 0, 0, 0] #Volos { thread_id: 371056, access_type: Read, locks_held: [5843256, 5843280] }
```

thread id contextualization

view of memory read/writes addresses + values

locks/futexes held

Figure: Volos race detection over an execution trace.

A credential check on an **untaken, input-gated branch**:

```
if opcode == 0x02 {           // opVerifyPeer
    getsockopt(fd, SO_PEERCREC) // CHECK
    ... race window ...
    readlink("/proc/<pid>/exe")  // USE
}
```

Overlay concolic execution flips the untaken branch, keeps the concrete input elsewhere, and *Z3 solves the opcode gate*: `arg1.byte_0 -> #x02 (opVerifyPeer)`

Two detection modes

- A. check-use-race**: both on-path; a thread switch in the window $\Rightarrow$ Critical.
- B. overlay-check-reachable**: the check is reachable only under attacker-influenced input; reported with its Z3-solved trigger.

The container-runtime TOCTOU line (CVE-2019-5736, Leaky Vessels) is exactly this shape in real Go code.

Backup: How to use Zorya to analyse a malware sample

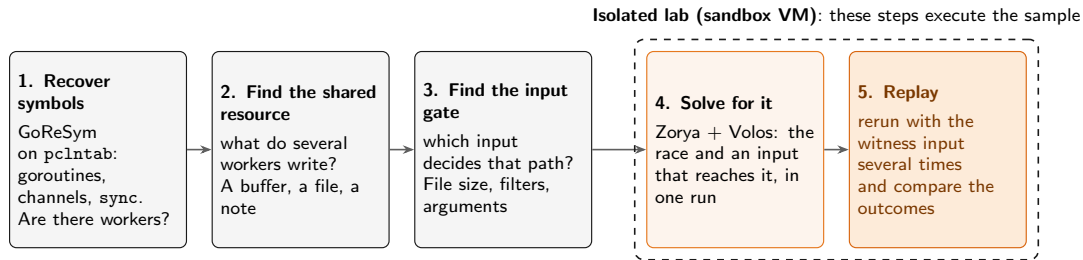


Figure: Analyst workflow for a malware sample: steps 4 and 5 execute the sample, in an isolated lab.

Steps 1 to 3 are manual today; step 4 is what Zorya and Volos do today.

Steps 4 and 5 were shown on models of VECT and DECAF, not on the samples.

Races that need a specific interleaving wait for the scheduler work listed in the perspectives.

Backup: Recovering function names from Go malware

Go keeps the `pcIntab` (Program Counter Line Table) to name functions in stack traces.

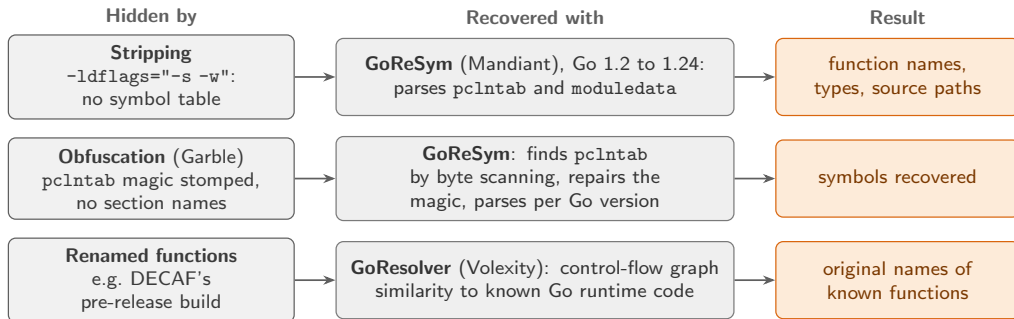


Figure: Recovering function names: what hides them, the tool that recovers them, and the result.

Function names stay recoverable after stripping, and often after obfuscation.

Source: Mandiant [49], Garble [50], Volexity [51], Morphisec [36] (DECAF); our test on a stripped Go 1.26 binary.

Backup: why C++ binaries are out of reach today

Test: `crash()` dereferences `v[argv[1][0] == 'K']`, `v = {valid, nullptr}`; natively, K segfaults. Zorya runs from the safe input a.

Build	What Zorya does
PIE, dynamic (g++ default)	stops at the stack-canary read <code>%fs:0x28</code> : no PIE relocation, no FS base
<code>-no-pie, dynamic</code>	false NULL-store report: library calls return 0, so operator <code>new</code> gives NULL
<code>-no-pie -static</code>	<code>strlen</code> is a skipped stub: <code>std::string</code> copies 0 bytes, the input is lost; then stops at <code>fstat (std::cout)</code>

Missing: exceptions and unwinding (`_cxa_throw`, `_Unwind_Resume`), demangled names, C++ tests.

Works only if the input goes from `argv` straight into user-code branches, without `std::string`, `new` or `iostream`.

The honest answer: I don't know.

Seen in the wild

VECT 2.0 (file size)
DECAF (file filter)

Not measured

two samples show the
pattern exists, not how
often it occurs

What it would take

a corpus of concurrent
malware, and a dataset
labelled on both axes

Quantifying this correlation is **a full piece of work of its own.**

Backup: The deliberate version: a hypothesis

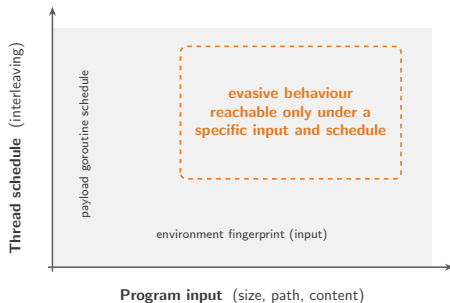


Figure: The deliberate version: evasive behaviour hidden behind both an input and a schedule.

What if the input $\times$ schedule coupling were not a bug but a **feature**?

Evasive code could hide a payload behind both.

It would have the **same structure** as VECT's accident, so the same tools could reach it.

We have not observed this in the wild.

- [1] Google.
The Go Programming Language, 2025.
- [2] Nghia Nguyen.
Go Scheduler, April 2025.
- [3] Nghia Nguyen.
Memory Allocation in Go, June 2025.
- [4] Securego.
GoSec : Golang static analyzer, 2024.
original-date: 2016-07-18T18:01:08Z.
- [5] Google.
GoVet : Golang static analyzer, 2024.
- [6] Google.
StaticCheck : Golang static analyzer, 2018.
- [7] Kamil Kisiel.
kisielk/errcheck: checking for unchecked errors in Go code., 2024.
original-date: 2013-02-24T22:32:02Z.
- [8] CodeQL.
CodeQL, 2021.
- [9] Snyk.
snyk/cli, 2024.
original-date: 2015-10-30T11:36:00Z.

- [10] Security-Research-Labs.
srlabs/golib afl, February 2026.
original-date: 2025-04-01T15:13:33Z.
- [11] Leanovate.
leanovate/gopter: the GOfang Property TestER, 2024.
original-date: 2016-02-11T07:20:49Z.
- [12] Trail of Bits.
trailofbits/krf: a KernelSpace Randomized Faulter., 2024.
original-date: 2018-12-16T20:28:15Z.
- [13] 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.
- [14] Sonal Mahajan.
NilAway: Practical Nil Panic Detection for Go, November 2023.
- [15] Netflix.
Chaos Monkey, May 2026.
original-date: 2016-10-18T22:36:41Z.
- [16] Long Zhang, Javier Ron, Benoit Baudry, and Martin Monperrus.
Chaos Engineering of Ethereum Blockchain Clients.
Distributed Ledger Technologies: Research and Practice, 2(3):1–18, 2023.

- [17] Benjamin Samuels.
Releasing the Attacknet: A new tool for finding bugs in blockchain nodes using chaos testing, March 2024.
Section: 2024.
- [18] 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.
- [19] Karolina Gorna, Nicolas Iooss, Yannick Seurin, and Rida Khatoun.
Zorya: Automated Concolic Execution of Single-Threaded Go Binaries, December 2025.
arXiv:2512.10799 [cs].
- [20] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R. Lyu.
Concolic Execution on Small-Size Binaries: Challenges and Empirical Study.
In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 181–188. IEEE/IFIP, June 2017.
- [21] Hui Xu, Zirui Zhao, Yangfan Zhou, and Michael R. Lyu.
Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs.
IEEE Transactions on Dependable and Secure Computing, 17(6):1243–1256, November 2020.
- [22] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan.
LAVA: Large-Scale Automated Vulnerability Addition.
In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, May 2016.
ISSN: 2375-1207.
- [23] Tim Boland and Paul E. Black.
Juliet 1.1 C/C++ and Java Test Suite.
Computer, 45(10):88–90, October 2012.

- [24] Kostya Serebryany.
OSS-Fuzz - Google's continuous fuzzing service for open source software.
2017.
- [25] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiyang Zhang.
Understanding Real-World Concurrency Bugs in Go.
In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 865–878, Providence RI USA, April 2019. ACM.
- [26] Check Point Research.
VECT: Ransomware by design, wiper by accident.
<https://research.checkpoint.com/2026/vect-ransomware-by-design-wiper-by-accident/>, 2026.
28 April 2026.
- [27] LevelBlue SpiderLabs.
Inside Vect ransomware-as-a-service.
<https://www.levelblue.com/blogs/spiderlabs-blog/inside-vect-ransomware-as-a-service>, 2026.
- [28] JUMPSEC Labs.
Bugs & betrayal: Vect analysis.
<https://www.jumpsec.com/guides/vect-ransomware-analysis/>, 2026.
- [29] Morphisec Labs.
VECT: Ransomware that can't decrypt.
<https://www.morphisec.com/blog/vect-ransomware-that-cant-decrypt/>, 2026.
- [30] Xinhao Yuan, Junfeng Yang, and Ronghui Gu.
Partial Order Aware Concurrency Sampling.
In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 317–335, Cham, 2018. Springer International Publishing.

- [31] Huan Zhao, Dylan Wolff, Umang Mathur, and Abhik Roychoudhury.
Selectively Uniform Concurrency Testing.
In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, pages 1003–1019, New York, NY, USA, March 2025. Association for Computing Machinery.
- [32] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte.
A randomized scheduler with probabilistic guarantees of finding bugs.
In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 167–178, New York, NY, USA, March 2010. Association for Computing Machinery.
- [33] Anmol Maurya.
Golang malware is more than a fad: Financial motivation drives adoption.
CrowdStrike blog, <https://www.crowdstrike.com/en-us/blog/financial-motivation-drives-golang-malware-adoption/>, 2021.
12 November 2021.
- [34] Stack-Overflow.
2025 Stack Overflow Developer Survey, 2025.
- [35] Intezer.
Year of the gopher: A 2020 Go malware round-up.
<https://intezer.com/blog/year-of-the-gopher-2020-go-malware-round-up>, 2021.
February 2021.
- [36] Hido Cohen and Michael Dereviashkin.
DECAF ransomware: A new Golang threat makes its appearance.
Morphisec Labs, <https://www.morphisec.com/blog/decaf-ransomware-a-new-golang-threat-makes-its-appearance/>, 2021.
28 October 2021.

- [37] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin.
DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools.
In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17*, New York, NY, USA, 2017. Association for Computing Machinery.
- [38] Paul Thomson, Alastair F. Donaldson, and Adam Betts.
Concurrency Testing Using Controlled Schedulers: An Empirical Study.
ACM Transactions on Parallel Computing, 2(4), 2016.
- [39] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue.
GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs.
In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 187–199, 2021.
- [40] Milind Chabbi and Murali Krishna Ramanathan.
A study of real-world data races in Golang.
In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI '22*, pages 474–489, New York, NY, USA, 2022. Association for Computing Machinery.
- [41] Ziheng Liu, Shihao Xia, Yu Liang, Linhai Song, and Hong Hu.
Who goes first? Detecting Go concurrency bugs via message reordering.
In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22*, New York, NY, USA, 2022. Association for Computing Machinery.
- [42] Joschua Schilling, Andreas Wendler, Philipp Görz, Nils Bars, Moritz Schloegel, and Thorsten Holz.
A binary-level thread sanitizer or why sanitizing on the binary level is hard.
In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1903–1920. USENIX Association, 2024.

- [43] Romain Pereira, George Stelle, and Patrick Carribault.
Taskgrind: Heavyweight dynamic binary instrumentation for parallel programs analysis.
In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 214–221. IEEE, 2024.
- [44] Xudong Sun, Zhuo Chen, Jingyang Shi, Yiyu Zhang, Peng Di, Jianhua Zhao, Xuandong Li, and Zhiqiang Zuo.
HardRace: A dynamic data race monitor for production use.
arXiv:2410.18412, 2024.
- [45] Fangming Gu, Qingli Guo, Lian Li, Zhiniang Peng, Wei Lin, Xiaobo Yang, and Xiaorui Gong.
COMRace: Detecting data race vulnerabilities in COM objects.
In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3019–3036. USENIX Association, 2022.
- [46] Ridhi Jain, Rahul Purandare, and Subodh Sharma.
BiRD: Race detection in software binaries under relaxed memory models.
ACM Transactions on Software Engineering and Methodology, 31(4):58:1–58:29, 2022.
- [47] Chongxin Zhong, Qidong Zhao, and Xu Liu.
BinGo: Pinpointing concurrency bugs in Go via binary analysis.
arXiv:2201.06753, 2022.
- [48] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang.
Detecting concurrency memory corruption vulnerabilities.
In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*, 2019.
- [49] Mandiant.
GoReSym: Go symbol recovery tool.
<https://github.com/mandiant/GoReSym>.



- [50] burrowers.
garble: obfuscate Go builds.
<https://github.com/burrowers/garble>.
- [51] Volexity.
GoResolver: using control-flow graph similarity to deobfuscate Go binaries.
<https://github.com/volexity/GoResolver>.