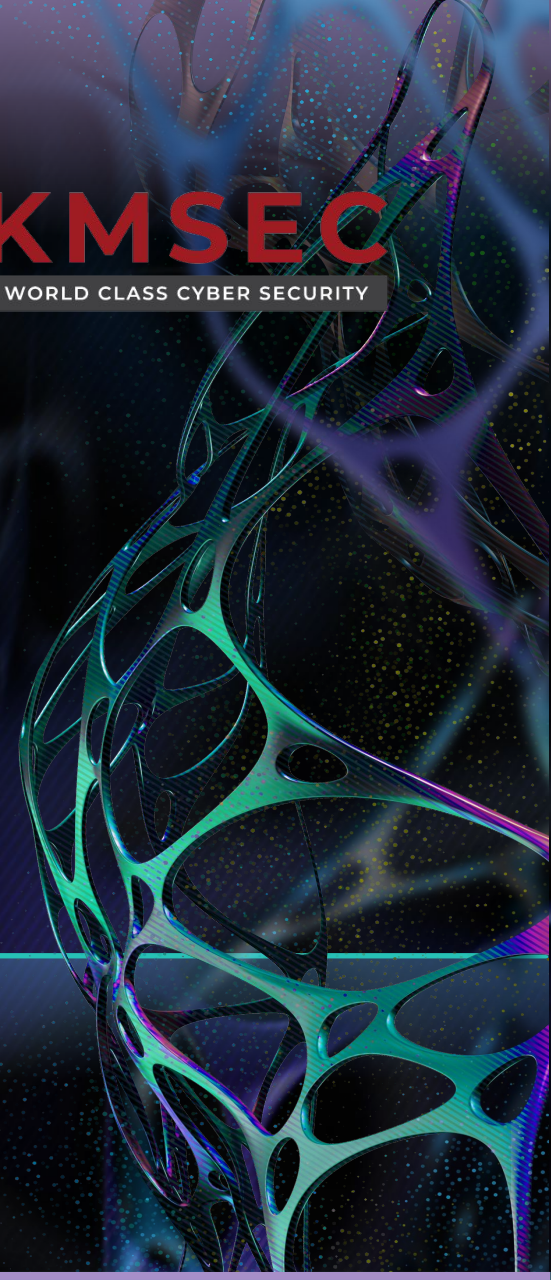

black hat®
ARSENAL
APRIL 23-24, 2026
MARINA BAY SANDS / SINGAPORE

black hat ASIA 2026

LEDGER DONJON

KMSEC
WORLD CLASS CYBER SECURITY



ZORYA

Go Binary Vulnerability Detection with Concolic Execution

Karolina GORNA

Software Security Engineer,
Ph.D. candidate,
Ledger Donjon and Telecom Paris

Keith MAKAN (MSc)

Founder + Principal Researcher,
KMSEC (PTY) LTD

Nicolas IOOSS

Security Software Engineer
Senior Staff
Ledger Donjon

Yannick SEURIN

Principal Security Cryptography Engineer
Ledger Donjon

Rida KHATOUN

Professor
Telecom Paris

1. ZORYA Concolic Executor Engine

An Overview of the Framework

Go language

- Go is a high-level programming language, **statically typed** and **compiled**.

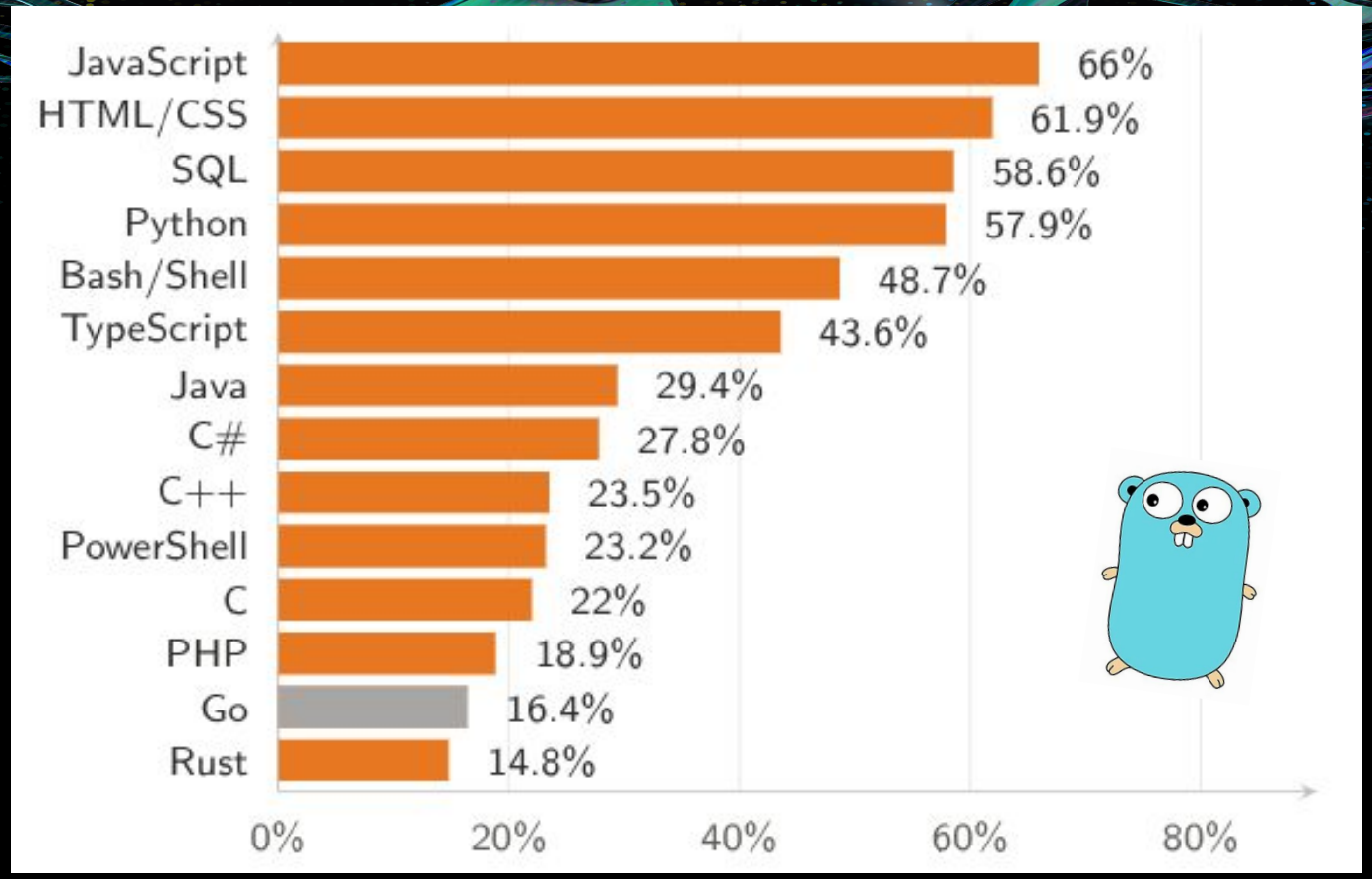


Figure: Most used programming languages among developers. Source: Stack Overflow Developer Survey 2025 (n = 31,771). Go ranks 13th with 16,4% of users

Go language

- *cmd/compile* produces goobj files under ABI Internal (register-based, since Go 1.17);
- *cmd/asm* uses the legacy stack-based ABI0;
- C code (dashed) is optional via cgo.
- *cmd/link* produces a staticallylinked native executable (ELF / Mach-O / PE) with the runtime embedded;
- This work focuses on ELF (Linux/x86-64)

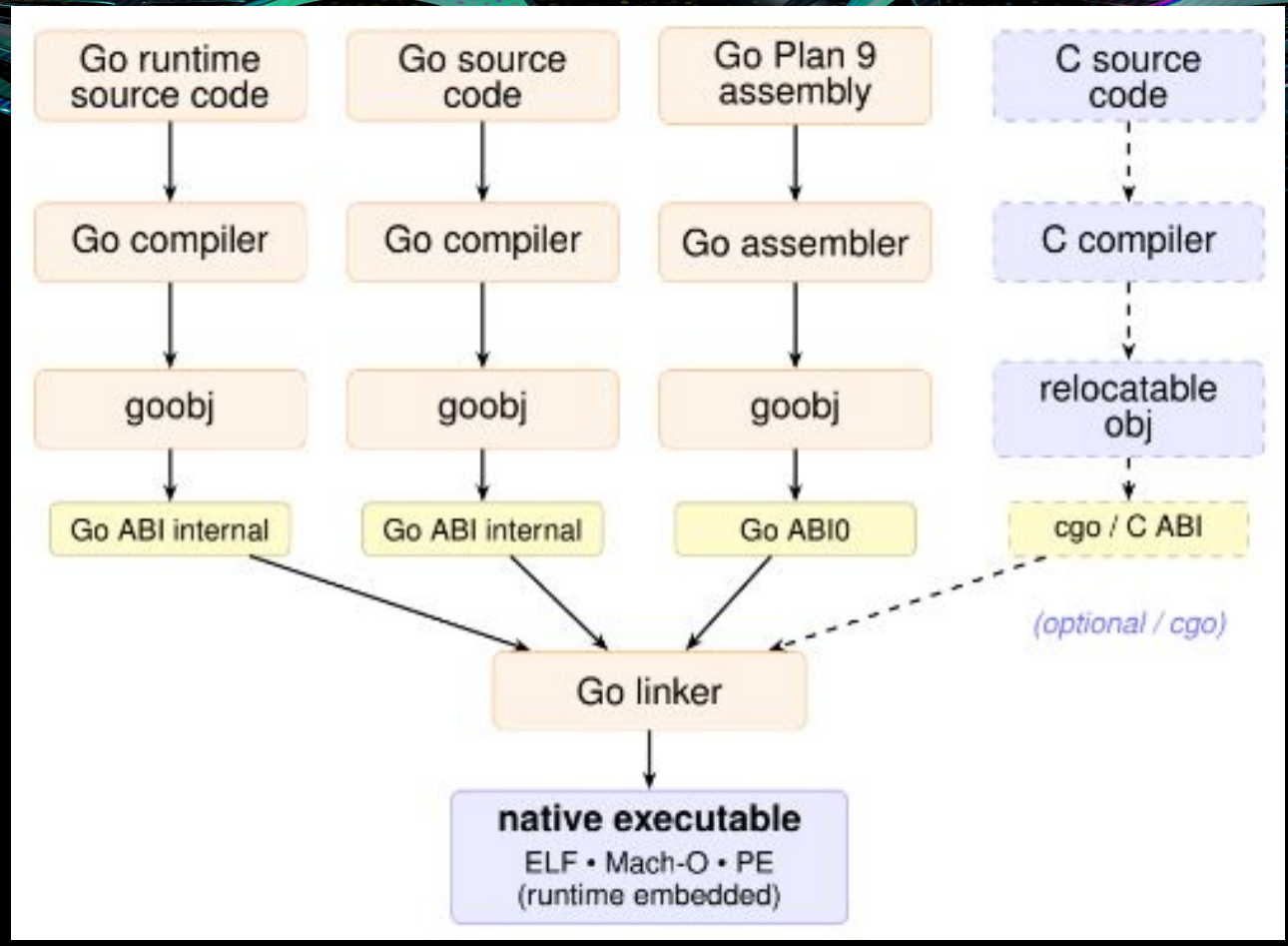


Figure: The Go compiler (gc) pipeline

Go Security Tool Landscape

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

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

Are there symbolic execution tools for Go?

Tool	Lan-guage	Method	Target	Intermediate Representa-tion (IR)	IR and Architecture Adapted for Go (limited / moderate / advanced)	Integrated SMT Solver(s)
Haybale	Rust	SE	Binaries	LLVM IR	No, gollvm is not maintained	Boolector
MAAT	C++	DSE	Binaries	P-Code	No, syscalls lacking support	Z3
KLEE	C++	SE	LLVM bitcode	LLVM IR	No, gollvm is not maintained	MetaSMT, STP and Z3
Owi	OCaml	Concolic Execution	Binaries	Custom IR	No, there is no direct compiler Go to WASM	Z3, Colibri2, Bitwuzla and CVC5
DuckEEGO	Go	Concolic Execution	Source Code	Go AST	Yes, limited	Z3
Radius2	Rust	SE + Taint Analysis	Binaries	ESIL	Yes, limited	Z3, Boolector
MIASM	Python	SE using CE + Binary Analysis	Binaries	Custom IR	Yes, limited	Z3, CVC4
BINSEC	OCaml	SE + concrete dumps	Binaries	DBA	Yes, limited	Z3, CVC4 and Boolector
Zorya	Rust	Concolic Execution	Binaries	P-Code and emulation	Yes, moderate	Z3

Figure: Comparison of existing symbolic tools with our methodology (SE:Symbolic Execution / DSE: Dynamic Symbolic Execution / CE: ConcreteExecution)

What is concolic execution ?

Static Analysis → Fuzzing → Symbolic Execution→ Formal Verification

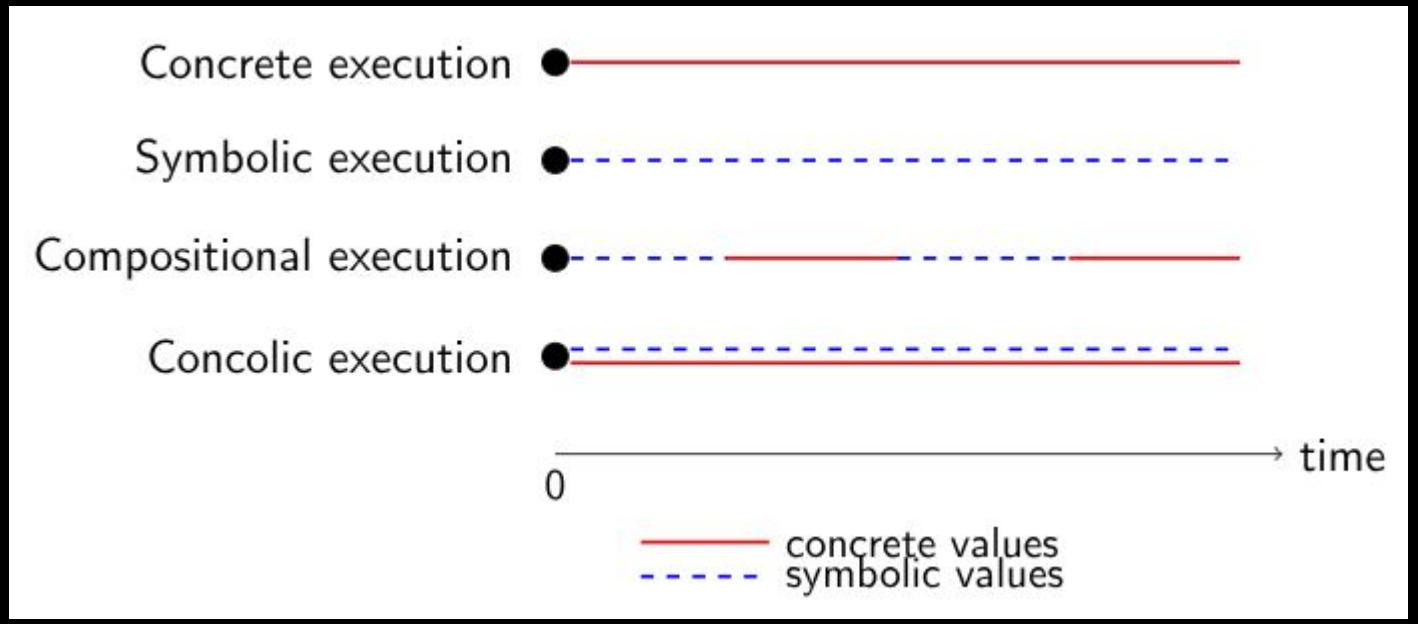


Figure: Different types of executions

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
                                          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: Example of low P-Code code from Ghidra

Zorya, pcode-generator, pcode-parser

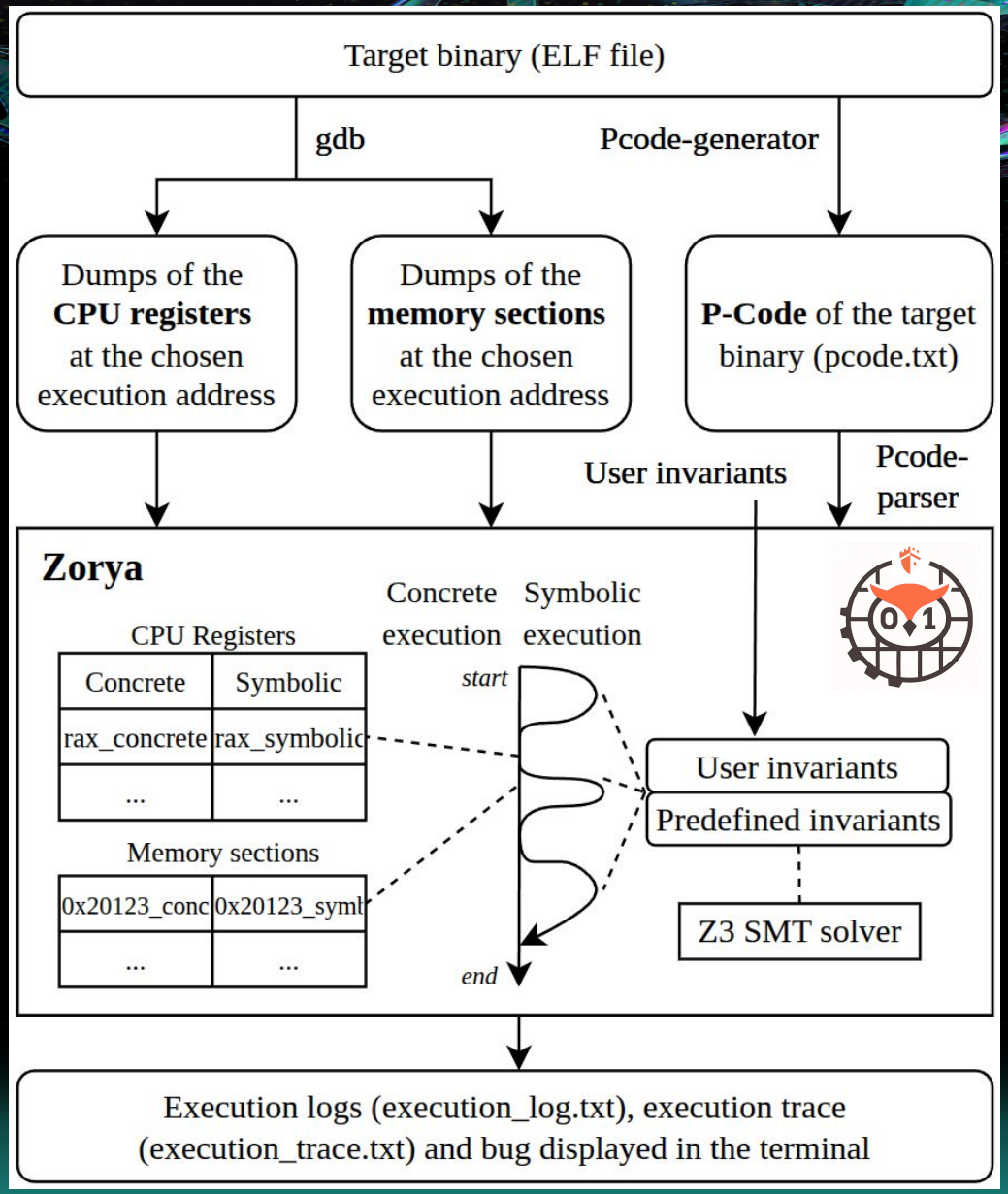


Figure: Zorya Execution workflow

Zorya exploration

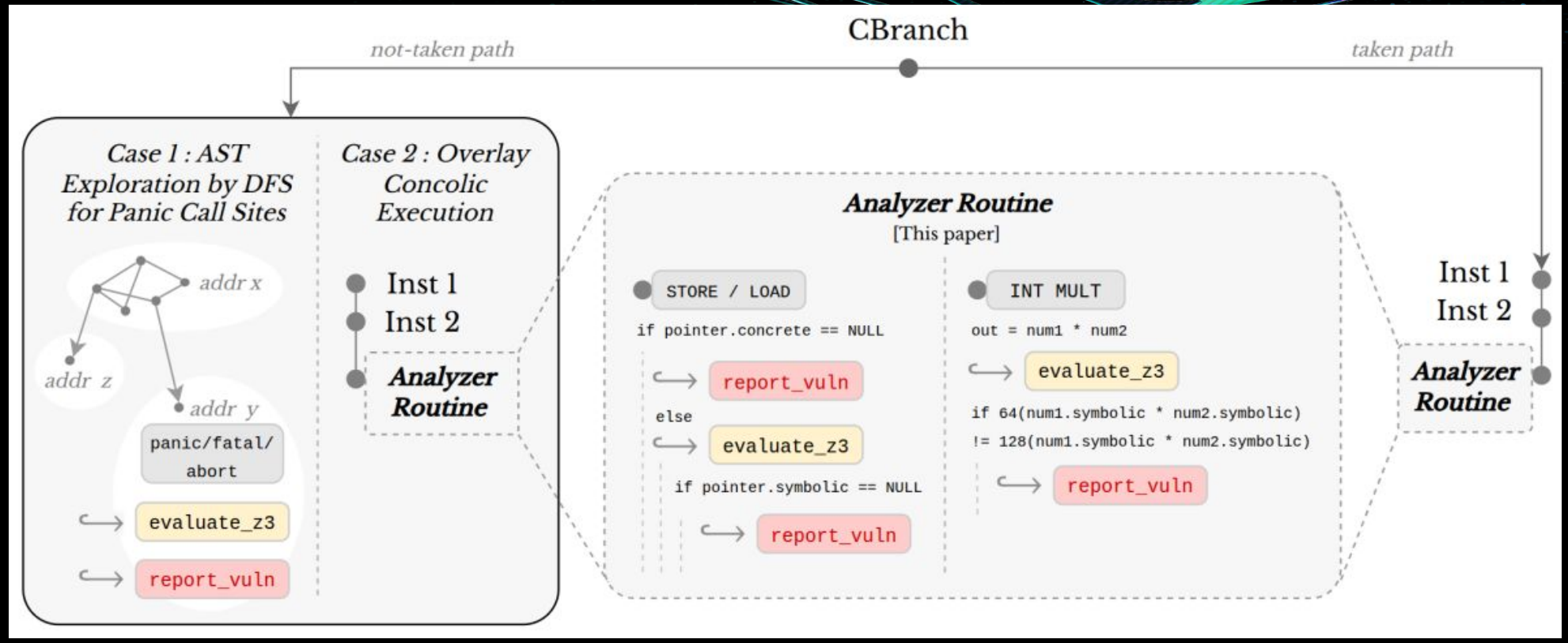


Figure: Overview of Zorya workflow, including AST exploration and Overlay Concolic Execution

Evaluation

		staticcheck	gosec	nilway	go -fuzz	GoLibAFL	Zorya	BINSEC	SymQEMU
Technique used		Static Anal.	Static Anal.	Static Anal.	Fuzzing	Fuzzing	SMT Solver	SMT Solver	SMT Solver
Needs Additional Files or Automated Execution (Auto.)?		Auto.	Auto.	Auto.	**Harness**	**Harness**	Auto.	**Init Files**	Auto.
Outputs the execution trace of each instruction?		No	No	No	No	No	**Yes**	**Yes**	No
Nil Pointer Dereference	kubectl-2025	.	.	.	✓	✓	✓	.	.
	kubelet-2025	.	.	.	✓	✓	✓	.	.
	geth-graphql-2025	.	.	✓	✓	✓	✓	.	.
	geth-tracers-2024	.	.	✓	✓	✓	✓	.	.
Integer Overflow	ep224-elliptic-2021	.	.	.	.	.	.	.	.
	fasthttp-2020	.	.	.	.	.	.	.	.
	tendermint-2018	.	.	.	.	.	.	.	.
	evm-gascost-2017	.	.	.	.	.	✓	.	.
Index Out of Bounds	kube-sm-2025	.	.	.	✓	✓	.	.	.
	coredns-2025	.	.	.	✓	✓	✓	.	.
	goprotobuf-2013	.	.	.	✓	✓	✓	.	.

Table: Evaluation of Go toolchain tools, symbolic execution tools and Zorya against a real-world Go bugs dataset. ✓ indicates caught vulnerability; empty cells indicate the vulnerability was not caught.

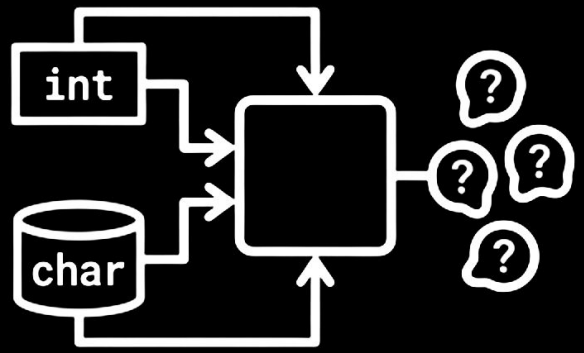
2. VOLOS Binary Race Condition Analysis

An Overview of the Architecture

Binaries are hard to analyze

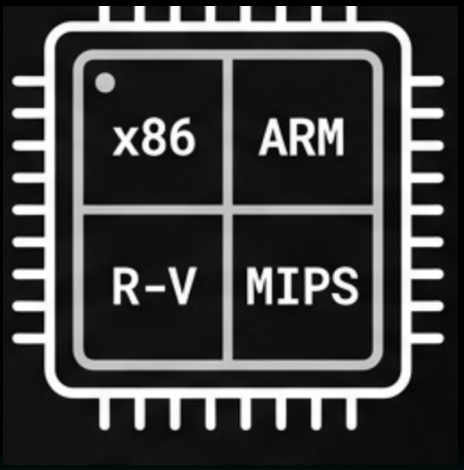
Variable Liveness / Data Flow

Loss of Typing Information

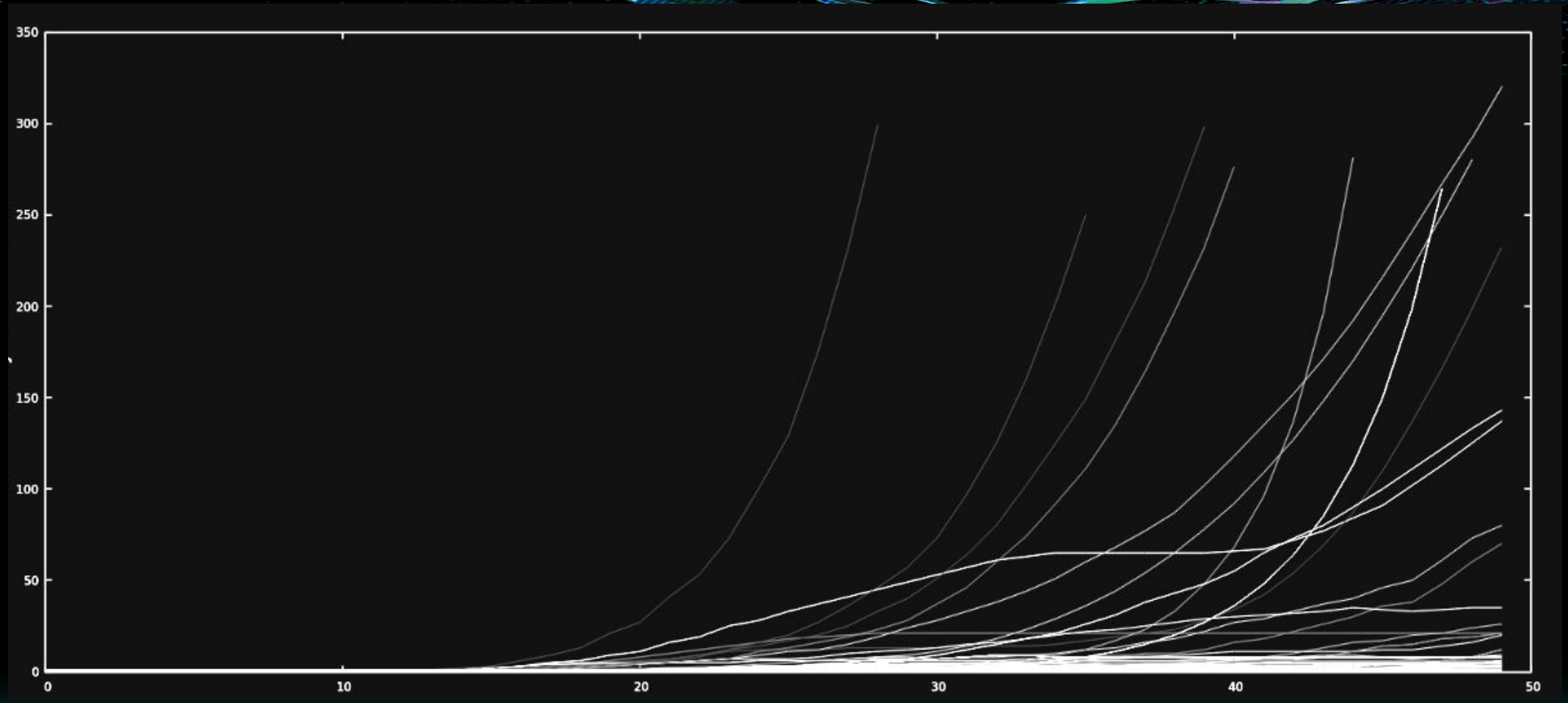


Loss of Semantic / Symbol Information

Multiple ISAs / Hardware Complexities



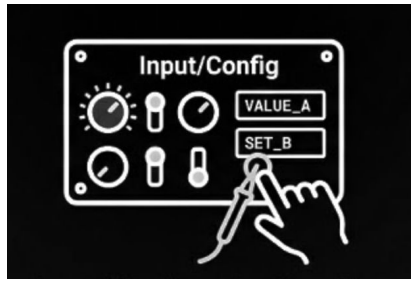
Symbolic Execution is hard to scale



Race Conditions are hard to detect

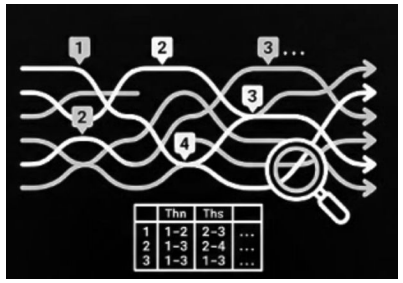
Shared Resource Analysis

Channels, sockets files, shared memory



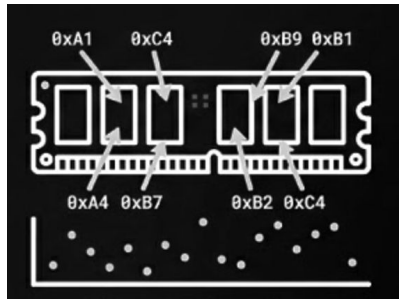
Input / Config Sampling

Finding samples that trigger thread activity



Thread Interleaving Analysis

Threads spawn in different schedulings, reorder, die and respawn



Memory Interaction Sampling

Sample ahead of time? during execution? analyze after execution?

Adapting Scheduling to Concurrency Bugs

Checkpoint type	Bugs exposed ?	Why this matters?	Which strategy is better?
Futex syscall	1. Deadlocks 2. Order violations on shared states	Thread is about to block or wake up, so possible issues	1. FIFO Strategy (Ensures all threads get fair chances to acquire locks)
Clone syscall	3. Initialization races 4. Parent-child data races	Parent/child interleaving determines if initialization happens before use.	3. Child-First Strategy (Forces child to run immediately, exposing use-before-init)
Memory interactions (atomic ops)	5. Data races 6. Atomicity violations	Other threads might see inconsistent state.	5. Parent-Child Locality (Parent and child often share memory. Switching between them at memory barriers exposes racy accesses) 6. Preemption at Memory Barriers (Switching between check and update exposes atomicity breaks. Requires checkpoint between operations)

Zorya-Volos vs the World

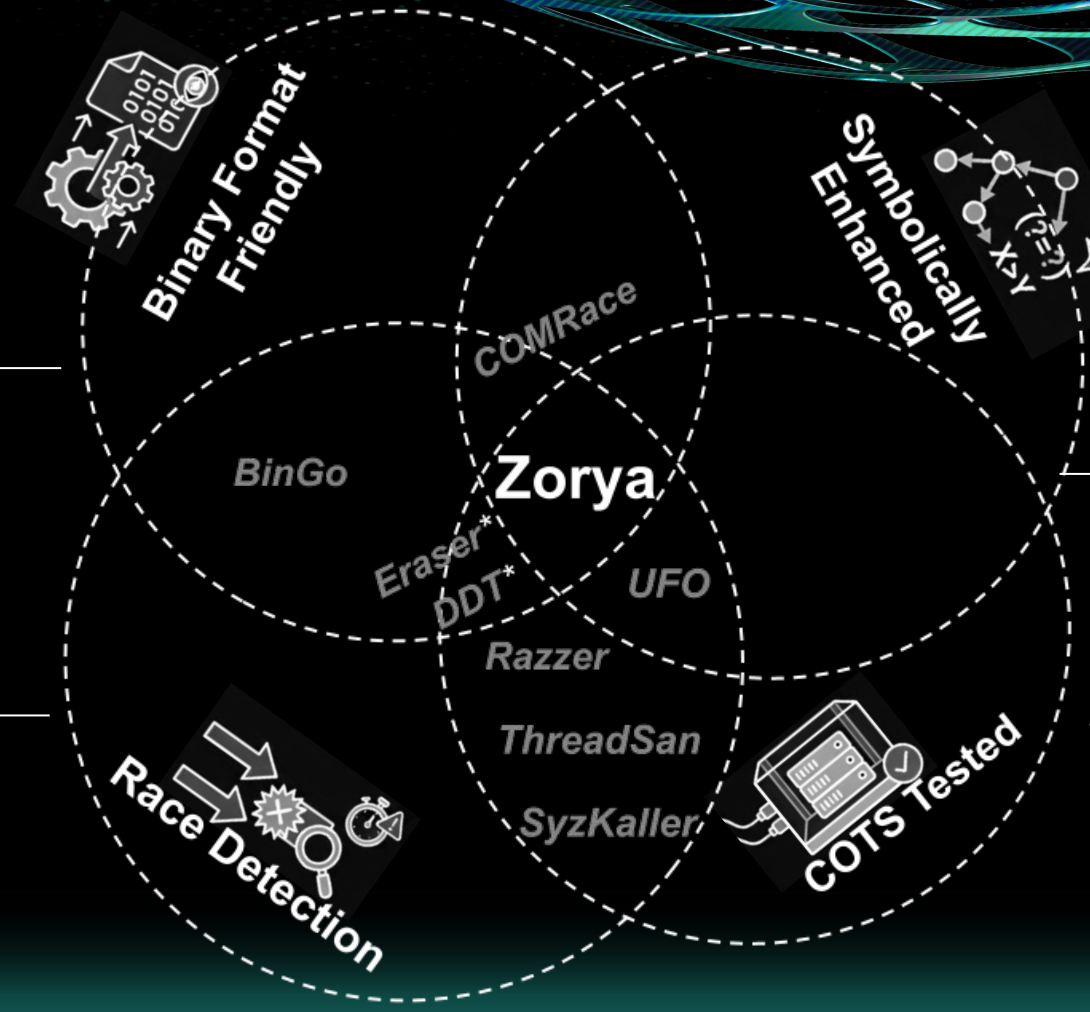
analysis without source code required!

automated test case generation

reproduces input samples

find real reproducible race conditions

tested on real world binaries



Binary Friendly Analysis Approaches are rare

Study	Target	Year	Analysis	RCDS	BFF	LSIP	COTS	Symbolic
Zorya	**Go**	**2026**	**LSA**/HBA	OTF	**Yes**	**Yes**	**Yes**	**Yes**
BinTSAN	**C/C++**	2024	LSA/HBA	OTF	**Yes**	Yes	**Yes**	No
TaskGrind	**C/C++**	2024	HBA	PM	**Yes**	No	**Yes**	No
HardRace	**C/C++**	2024	LSA/HBA	PM	**Yes***	**Yes**	**Yes**	**Yes**
COMRace	**C/C++**	2022	LSA	CT	**Yes**	No	**Yes**	**Yes**
BiRD	**C/C++**	2022	LSA/HBA	PM	**Yes**	No	No	No
BinGo	**Go**	2022	LSA	PM	**Yes**	No	**Yes**	No
ConVul	**C/C++**	2019	HBA	OTF	**Yes**	**Yes**	**Yes**	**Yes**

Table: Contemporary approaches to binary race condition detection. RCDS (‘Race Detection Style’) indicates if detection happens On-The-Fly, during execution, Post-Mortem, after execution or CT at compile time. BFF (‘Binary Format Friendly’) indicates it accepts binary/executables as targets.

Happens-Before + LockSet Analysis

Code

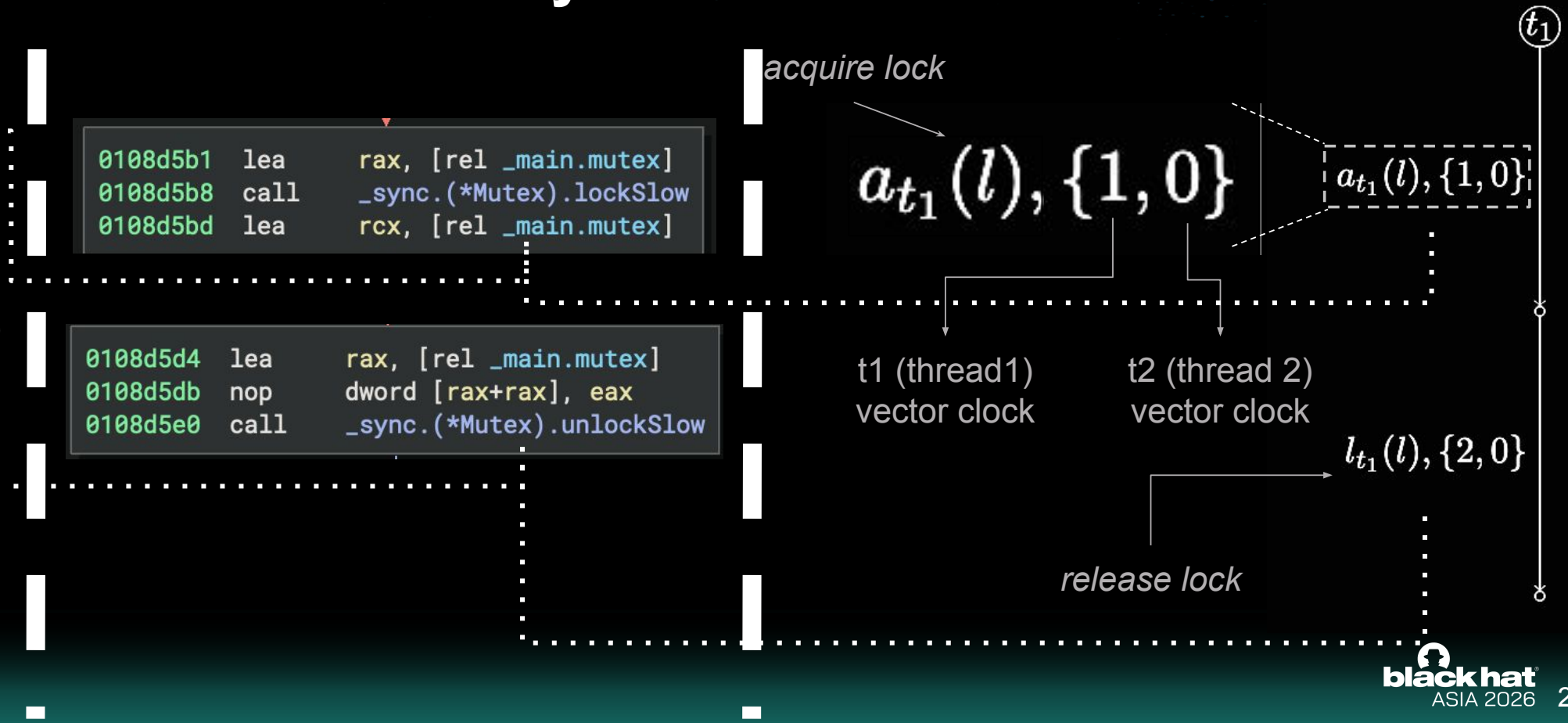
```
go func() {
  mutex.Lock()
  ...

  mutex.Unlock()
}()
```

Disassembly

```
0108d5b1  lea     rax, [rel _main.mutex]
0108d5b8  call    _sync.(*Mutex).lockSlow
0108d5bd  lea     rcx, [rel _main.mutex]
```

```
0108d5d4  lea     rax, [rel _main.mutex]
0108d5db  nop     dword [rax+rax], eax
0108d5e0  call    _sync.(*Mutex).unlockSlow
```

Release / acquire event



Happens-Before + LockSet Analysis

Code

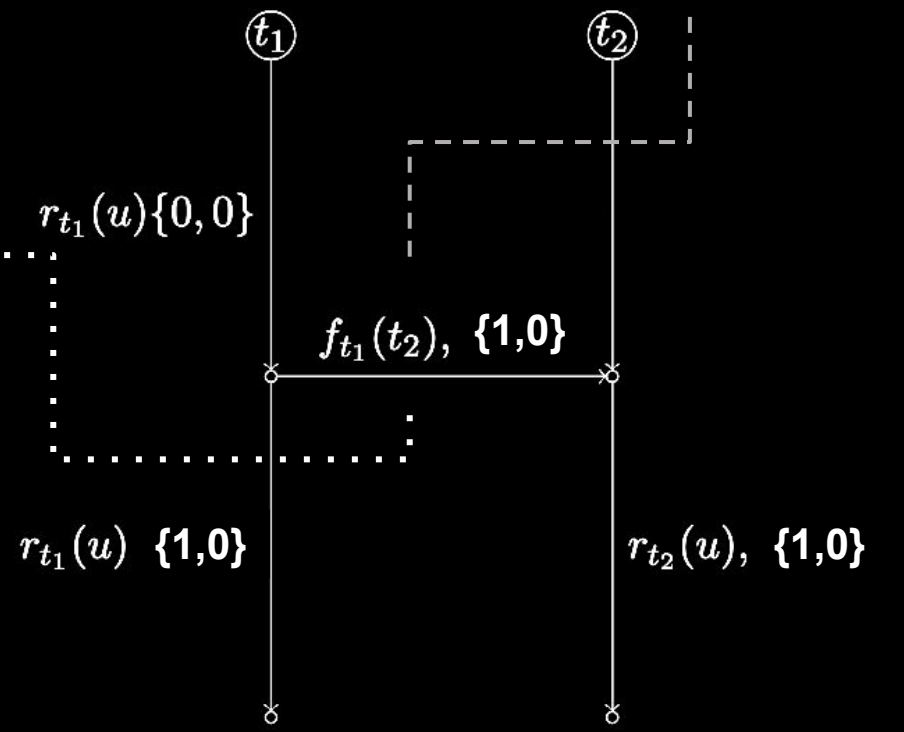
```
func main() {
    // Main Thread
...
    // Child Routine 1
    go func() {
...
    }()
...
    // Child Routine 2
    go func() {
```

Disassembly

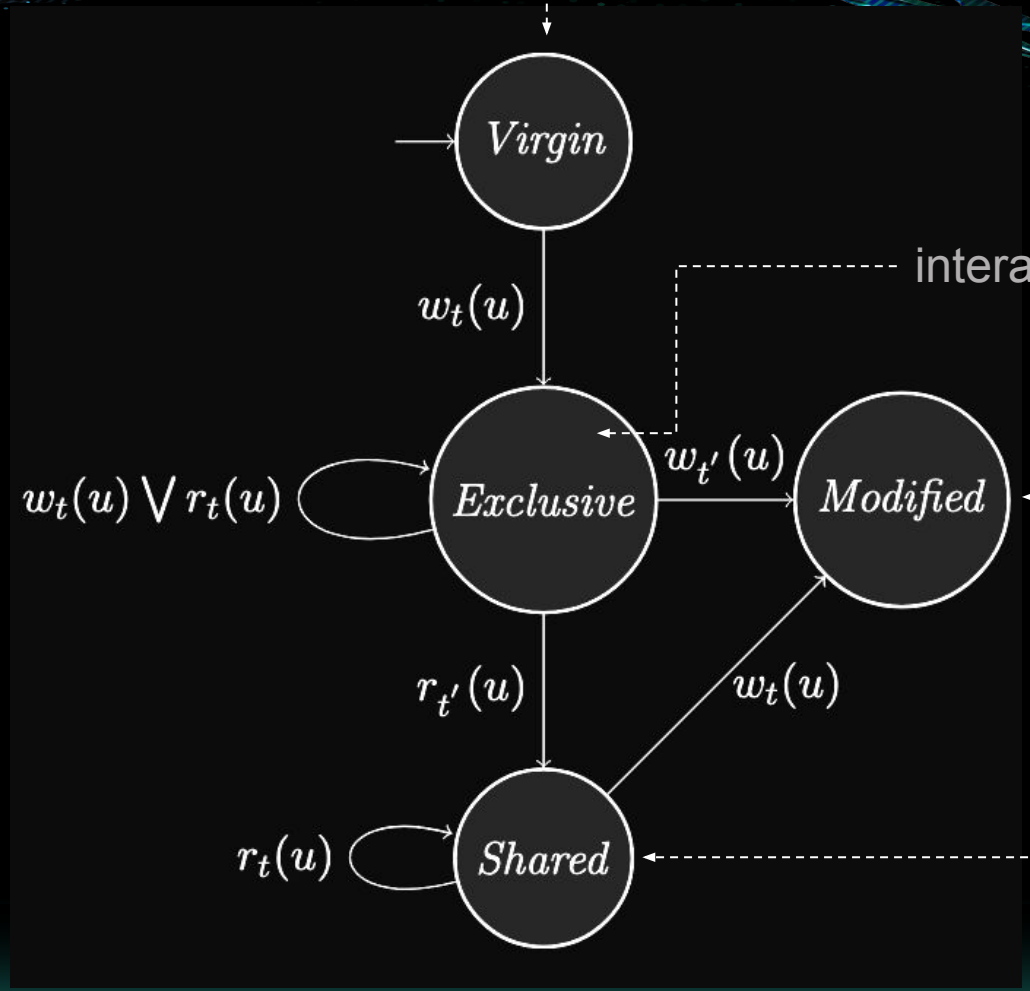
```
0108d5e5  lea     rax, [rel data_10ade10]
0108d5ec  call    _runtime.newproc
0108d5f1  lea     rax, [rel data_10ade18]
0108d5f8  call    _runtime.newproc
```

Fork/Clone event

fork new routine/thread



Happens-Before + LockSet Analysis



no interaction

interaction in one thread

more than one thread interaction

race potential!

Volos in action

```
[VOLOS] WRITE MEM @[0x7fffffffd780  ] <= a5 86 4(        ): 0x0, vector_clock: VolosVC { node_id: "0", clocks: {"0": 0} } }
Address: 448700, Symbol: runtime.newproc1 -> fn=(        :0x0 (reg=RSI @0x30), ~r0=0x0 (reg=R9 @0x88)

[VOLOS] Tick for runtime.newproc1 (TID: 3840856)(

[VOLOS] READ  MEM @[0x56eb60        ] <= e0 dd 5(        vector_clock: VolosVC { node_id: "0", clocks: {"0": 0, "3840856": 1} } }
[VOLOS] READ  MEM @[0x56de78        ] <= 00 00 0(        vector_clock: VolosVC { node_id: "0", clocks: {"0": 0, "3840856": 1} } }
```

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

Volos in action

```
------------------------------------------------------------
[VOLOS DETECTOR] DATA RACE FOUND AT 0x1f955
------------------------------------------------------------
Access 1:
  Goroutine ID: 284583
  Op Type:      Write
  Locks Held:   [5843280]
--- VS ---
Access 2:
  Goroutine ID: 284586
  Op Type:      Write
  Locks Held:   NONE (UNPROTECTED)
------------------------------------------------------------
REASON: One or more threads accessed this memory without a shared lock.
```

ZORYA - Concolic Execution Engine for Go Binaries

zorya.karolinagorna.net

	Feature	Functional	Scaled	Notes
1	Comparison to BinGo, others	✓	✗	*Currently building a more contemporary dataset, collecting data*
2	Go Routine Localization	✓	✗	*Reporting with goroutine context included ~ 90% complete*
3	Ad Hoc Location analysis	✓	✗	*Can currently execute from a function boundary*
4	Adaptive response to LSIP	✓	✗	*Detects races were symbols are available, adaptive methods soon!*
5	POR Thread Interleaving Enumeration	✓	✗	*Currently naive, brute force, Partial Order Reduction (POR) techniques soon to be tested.*

LEDGER DONJON

KMSEC
WORLD CLASS CYBER SECURITY

Thank you!

zorya.karolinagorna.net

Questions ?