E0: Selective Symbolic Instrumentation

Powering Data-Flow Fuzzing and LLM Reasoning

AUTHOR: Jex Amro
@jexamro <jx@squarelabs.ai>

Table of Contents

Introduction

Imagine a fuzzer driven by data-flow rather than code-coverage. One that follows input def-use chains from source to sink, enforces constraints at each step, and reasons only about the precise, ordered low-level operations that govern data propagation. Instead of blind randomness, it employs a mutation engine guided by symbolic reasoning and high-level semantic context.

Initially, LLMs remain in deep sleep, awakened only when their broader insight is needed to analyze complex vulnerabilities requiring higher-level semantic or cross-function reasoning. You can query this engine to inspect its findings, direct its focus, and steer exploration toward areas of interest.

We call this multi-layered binary analysis and instrumentation framework E0 (Explore from Ground Zero). E0 integrates fuzzing's speed, selective symbolic instrumentation's precision, and LLM guidance's semantic understanding, leveraging each technique's strengths and minimizing their limitations.

While a full open-source release of the framework is planned, this Phrack issue includes an alpha snapshot of the most critical components supporting the techniques discussed here, enabling security researchers and engineers to directly experiment with and validate our novel methods.

In this paper, we outline the challenges encountered in building E0 and introduce the solutions we developed.

1 - Fuzzing

It is mid-2025 and fuzzing remains the cornerstone of automated vulnerability discovery, thanks to its ability to generate massive numbers of diverse inputs at high speed, and to produce concrete, reproducible failures (crashes, hangs, and memory violations) that demand developer attention. By randomly mutating inputs guided by simple coverage heuristics, modern fuzzers can traverse thousands of execution paths per second with virtually no manual effort. However, fuzzing's randomized nature and its over-reliance on code-coverage metrics often leave deep, tightly guarded branches unexplored and miss subtle logic flaws or boundary conditions. Furthermore, without any semantic understanding of programstate, fuzz campaigns can spend inordinate resources on redundant or low-value paths, making it difficult to target high-impact code regions without massive computational investment.

2 - Symbolic Execution

Symbolic execution fills fuzzing's blind spots by treating inputs as symbolic variables that are transformed into logical formulas and systematically exploring program paths via constraint solving. Its strength lies in generating concrete test cases for specific branches, uncovering corner-case errors, assertion failures, and subtle boundary violations without relying on coverage heuristics. For example, in a nested buffer-copy loop it can derive an input that triggers an off-by-one overflow or calculate the exact values needed to cause an integer overflow, or solve for a memory write that lands at a specific address.

Yet symbolic execution operates only at the level of individual instructions and their accumulated formulas; it has no built-in notion of buffer sizes, object lifetimes, or business-logic rules. Without manually supplied checks, such as boundary conditions, function summaries, loop invariants, or custom heap models, many critical bugs go undetected. These include out-of-bounds writes, use-after-free conditions, and missing validations across multi-step transactions. Detecting a use-after-free, for example, requires encoding both allocation and deallocation semantics by hand, since the solver cannot distinguish freed memory from valid regions.

This instruction-centric view also leads to explosive path growth: each branch doubles the search space and quickly overwhelms time and memory resources on complex binaries. SMT solvers may timeout on large complex expressions, and external interactions such as network I/O, system calls, or threading often require hand-written stubs or rough approximations that can miss behaviors or introduce false negatives or positives. While strategies such as search heuristics (depth-first versus breadth-first), state merging, interpolation, constraint caching, and mixed concrete-symbolic (Concolic) runs can alleviate some of the limitations, the core challenges of scalability, and broader contextual understanding remain.

3 - Large Language Models (LLMs)

By contrast, Large Language Models bring complementary strengths by naturally absorbing and manipulating broader semantic layers (functions, modules, design patterns), and synthesizing intent across

thousands of lines of code. For example, given a snippet like:

```
char buf[16];
write_user_data(buf, user_len);
```

LLMs can warn: "Potential buffer overflow if 'user_len' > 16," without requiring any additional boundary specifications.

In another snippet:

```
ptr = malloc(...);
free(ptr);
ptr->field = x;
```

LLMs can flag a use-after-free issue: "You're writing to freed memory here; ensure you don't access 'ptr' after it's freed." They can also spot missing business invariants, such as failing to roll back a transaction on failure, or suggest protocol fixes that span multiple functions.

Yet they still stumble over low-level assembly and raw binaries: they hallucinate flag conditions ("this jump tests zero" when it actually tests carry), cannot compute dynamic branch targets, and hit context-window limits on very large repositories (truncating critical paths in a 20,000-line module). They struggle to pinpoint exact lines of code and visualize deeply nested jump chains, especially when registers are reused to hold different variables, which distracts their token-based attention.

Time and computational cost further constrain them: longer contexts incur higher inference latency and significant compute expenses, making exhaustive analysis of massive code bases impractical. Despite these drawbacks, by leveraging prompt engineering, fine-tuning on domain-specific code, Model Context Protocol (MCP), and retrieval-augmented generation, researchers can still extract high-impact insights into memory-safety flaws, race conditions, and business-logic errors.

However, LLMs frequently generate a high volume of false positives during vulnerability analysis when they lack precise data-flow and constraint information. In practice, an LLM may flag any external input as a potential attack surface, even when the input cannot reach or influence the vulnerable code in question. This over-flagging forces researchers to manually review numerous false findings or to invoke additional LLM inference rounds, sometimes with narrower prompts or more compute-intensive techniques, to validate true positives. While LLMs can deliver accurate results when a vulnerability is fully contained within a single function or module, real-world vulnerabilities often span multiple functions, modules, or processes. Reconstructing and supplying the full cross-function context is both challenging and expensive, in terms of engineering effort and inference latency, making validation and false-positive reduction the primary hurdles in applying LLMs to large-scale binary vulnerability research.

4 - Towards a Hybrid Approach

The contrast between symbolic execution's instruction-by-instruction precision and LLMs' broad, high-level reasoning highlights how they can address each other's blind spots. Symbolic execution supplies exact data-flow semantics: accumulated path constraints; variable liveness; input reachability; solved constraint-variable ranges; and cross-thread flows directly into AI pipelines, sparing LLMs the heavy lifting of inferring low-level data-flow details across vast source-code or decompiled binaries and thus dramatically reducing the volume of false positives that undermine purely LLM-driven analyses. In addition, since compiler optimizations and decompilation can alter or obscure actual control-flow and binary behavior, relying solely on source or decompiled code risks missing hidden bugs, whereas symbolic slicing operates on the real binary, preserving true execution behavior.

In contrast, LLMs inject domain context, established design patterns, and high-level business-logic checks that symbolic tools cannot model. Combined in a dynamic feedback loop (fuzzing, on-demand symbolic slicing, and LLM-guided review), you achieve a self-reinforcing pipeline: fuzzers explore new paths informed by solved constraints; symbolic analysis attaches precise data-flow invariants; and LLMs transform those invariants into actionable, human-readable insights, all validated through native binary execution.

An ideal hybrid system harnesses the raw throughput of unmodified binaries; triggers symbolic emulation only on input-derived code regions for minimal overhead; and enriches each LLM prompt with concise slices scoped to one or more functions (including their constraint inputs: arguments, memory-load values, and call-return values), augmented by SMT-solved variable ranges to eliminate infeasible scenarios up front.

This fusion arms LLMs with concrete, high-fidelity context; reduces wasted inference cycles on false positives; and closes the loop with end-to-end fuzzing, uniting discovery breadth, symbolic precision, and AI reasoning into one scalable, low-latency vulnerability-discovery engine.

4.1 - Data-Flow vs Code-Coverage Guidance

Data-Flow Guidance centers input exploration on the precise propagation of attacker-controlled data through program operations. By instrumenting and tracking def-use chains (capturing only those instructions that carry data derived from guarded inputs), E0 ensures that mutations and analyses focus exclusively on branches and operations aligned with data-flow. In contrast, Code-Coverage Guidance simply chases new basic blocks or branch targets without regard to semantic relevance. It may exercise the same edges thousands or even millions of times, wasting fuzz cycles on paths that bear no relation to exploitable state. Its only advantage is raw throughput: high volumes of test cases with minimal semantic insight. Data-Flow Guidance reduces noise, yields higher-quality test cases in fewer iterations, and in our approach, supplies precise data-flow context for downstream symbolic and AI layers to perform deeper reasoning, predict and help discover vulnerabilities, and generate inputs that trigger them. Moreover, unlike Code-Coverage Guidance (which requires its monitoring scope to be tied to specific modules or regions), data-flow instrumentation via (SSI) Selective Symbolic Instrumentation is applicable across all modules and threads (with optional scope narrowing when needed), enabling E0 to track untrusted-input data-flow throughout the entire application without sacrificing performance.

4.2 - E0 Design Decisions

After extensive research and development, and multiple full rewrites, E0's design has converged on several core principles:

- Harness-Based Fuzzer Compatibility: E0 operates exactly like a modern fuzzer by executing a user-supplied harness iteratively. Users provide a harness for the target functionality, along with a module name, a relative virtual address, and a register to capture, mark, and guard inputs. On each iteration, E0 takes the current sample and spawns new samples for every newly solved path constraint. With the optional -x flag, E0 will recursively execute each newly generated sample to further expand its exploration.

- Crash-and-Hang Detection (in typical fuzzing fashion): the classic "definite outcome" mechanism, where E0 drives your harness and watches for native crashes, hangs, or illegal-memory accesses (even catching them early in emulation mode before they'd be swallowed by OS exception handlers).

- Decoupled AI Analysis: To maximize selective symbolic instrumentation performance, E0 separates the heavy lifting of data-flow tracking from downstream AI processing. When run with the -log_symdefs flag, E0 writes collected symbolic definitions into a SQLite database, which AI agents can consume asynchronously in a separate process.

- AI-Driven Prediction: symbolic data is fed to LLMs that flag likely vulnerabilities ahead of an actual trigger, and in the near future, will loop back to the fuzzer to validate generated candidate inputs.

- Selective Instrumentation: We combine hardware watchpoints with OS-level page guards, then fall back exclusively on recursive, bitmap-driven watchpoint allocation for minimal overhead and maximal precision.

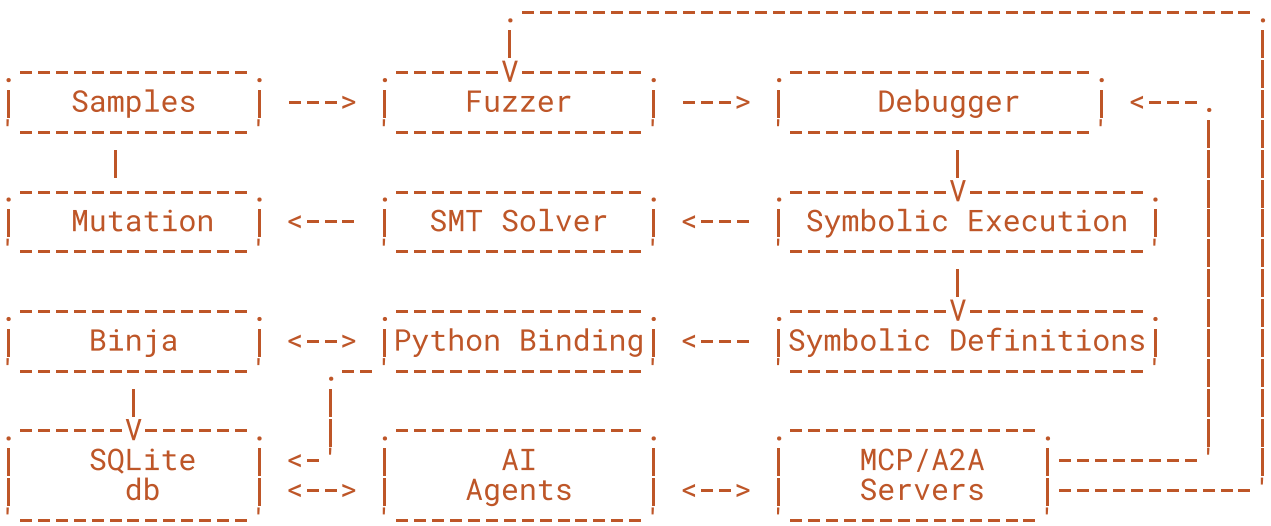
- On-Demand Emulation: By using Triton's ONLY_ON_SYMBOLIZED mode, symbolic analysis is invoked only when a guarded memory access introduces a new symbolic operand, then immediately reverts to native execution once all symbolic state is concretized.

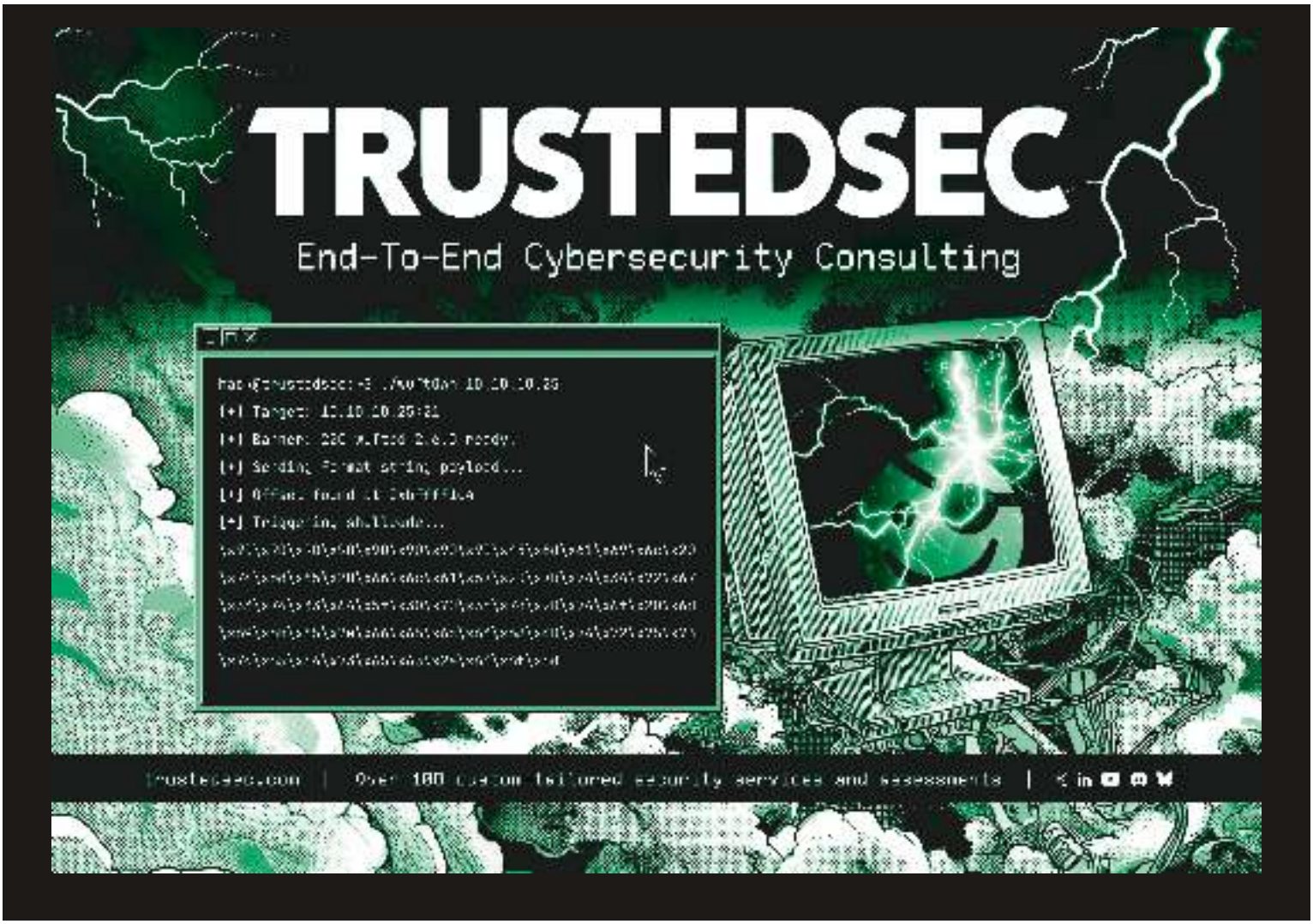
- Layered Integration: Fuzzing provides breadth, the SMT solver refines precise path constraints, and an LLM-driven AI layer supplies high-level semantic reasoning. This synergy balances speed, accuracy, and contextual insight.

These decisions enable the framework to operate from ground zero on closed-source binaries, scale across large, multithreaded codebases, and supply rich low-level symbolic insights to the AI layer, paving the way for high-confidence, low-overhead vulnerability discovery.

5 - E0 Architecture

E0 is architected as a multi-layered framework in which each component contributes to an integrated vulnerability discovery process.





E0: Selective Symbolic Instrumentation

[1] - Fuzzer

Role: Drives input generation and verifies vulnerabilities.

Function: Iteratively produces new inputs guided by solutions from the SMT solver, targeting unexplored or under-explored paths to increase overall path exploration and the likelihood of triggering vulnerabilities. In addition, the fuzzer plays a role in vulnerability verification by reproducing triggering conditions and validating observable failures such as crashes or hangs.

[2] - Dynamic Instrumentation (Debugger - LiteTracer)

Role: Manages the seamless switching between native execution and symbolic emulation.

Function: Monitors runtime execution using hardware watchpoints and protected memory pages. It triggers Targeted Emulation when a memory access on a traced guarded memory location is detected, ensuring that symbolic analysis is focused on relevant code paths.

3] - Targeted Emulation (Selective Symbolic Execution)

Role: Performs symbolic analysis only when necessary and manages the transition back to native execution.

Function: Utilizes techniques such as Triton's ONLY_ON_SYMBOLIZED mode to process only instructions involving symbolic operands. Importantly, the term "Selective Symbolic Execution" here encompasses not only the optimization provided by Triton's ONLY_ON_SYMBOLIZED mode but also the overall targeted emulation strategy initiated by the dynamic selective instrumentation layer (Section 6) when critical memory accesses are detected. In addition, this component is responsible for switching back to native execution once all symbolic registers are concretized, thereby minimizing emulation overhead while capturing detailed symbolic expressions that reflect critical program behavior.

[4] - SMT Solver (Z3 - Bitwuzla)

Role: Solves accumulated symbolic constraints.

Function: Computes precise variable ranges and refines symbolic expressions before passing them to the AI analysis layer, ensuring that the symbolic data is both accurate and actionable.

[5] - Binary IR and Decompiler (Binary Ninja)

Role: Provides detailed structural code context.

Function: Disassembles and decompiles binaries to extract function boundaries, signatures, assembly code, pseudocode, and intermediate representations (such as LLIL, MLIL, HLIL and SSA forms). These insights help contextualize the symbolic expressions for subsequent LLM analysis.

[6] - AI Layer (LLM Integration)

Role: Conducts vulnerability analysis.

Function: Leverages the well-structured symbolic expressions/definitions generated in the Selective Symbolic Execution layer (Section 6) to perform function-level or data-flow slices vulnerability assessments.

[7] - Model Context Protocol (MCP) & Agent-to-Agent (A2A)

Status: Early research and development.

Role: Context retrieval and action invocation.

Function: Provides AI agents with a unified interface to fetch analysis data (function code, solved constraints, symbolic definitions) to enrich LLM context on-demand, and to invoke E0 operations such as fuzzing, debugging, sample generation, and verification.

6 - Selective Symbolic Instrumentation (SSI)

Selective Symbolic Instrumentation tracks external inputs by installing targeted memory monitors. These monitors trigger symbolic analysis only when external inputs directly influence execution, minimizing unnecessary emulation overhead. Each symbolic slice precisely captures data-flow and control-flow semantics, producing structured constraints for downstream SMT solving and AI analysis. This ensures deep, meaningful analysis precisely where needed, while preserving native execution speed elsewhere.

The process begins by placing an OS-level guard on the input buffer's page. Any load from this guarded page generates a fault, pinpointing the first instruction to consume external bytes and spawning a focused symbolic emulation slice seeded with a concrete program state. As symbolic values propagate into registers, heap objects, stack locals, or globals, each newly touched page is likewise guarded. Subsequent faults cascade, mapping out the data-flow graph one step at a time as accumulated symbolic expressions, and ensuring that symbolic reasoning engages only when and where inputs actually affect control or state.

```
Example: Symbolic Emulation Session on a memmove Loop
memmove 0x1804ab304: stp q0, q1, [x3]  <- Symbolized mem write
memmove 0x1804ab308: add x3, x3, #0x20   // Not Symbolized
memmove 0x1804ab30c: ldnp q0, q1, [x1] <- triggers emulation
memmove 0x1804ab310: add x1, x1, #0x20   // Not Symbolized
memmove 0x1804ab314: subs x2, x2, #0x20  // Not Symbolized
memmove 0x1804ab318: b.hi #0x1804ab304   // Not Symbolized
```

In this example, emulation is triggered at the first 'ldnp' instruction (0x1804ab30c) when the guarded source memory is accessed. Symbolic execution starts using Triton's ONLY_ON_SYMBOLIZED mode. Emulating that 'ldnp' instruction results in q0 and q1 registers being symbolized. As the loop continues, subsequent 'add' and 'subs' instructions will be emulated by Triton but remain unmarked for data-flow since they do not operate on symbolized operands. At 0x1804ab304 'stp q0, q1, [x3]' Triton will symbolize the destination memory because the source registers are symbolized. A Triton memory-access callback, set by E0, is then triggered:

E0 guards the destination page, creating a new symbolic memory region.

Emulation proceeds until all symbolic registers have been concretized (i.e., no further symbolic registers dependencies remain - only symbolic memory), at which point emulation ends and all guards are lifted to allow the debugger to natively step over the emulated code. Native execution then resumes until the next guarded memory access triggers another symbolic emulation session.

The memmove example above represents a symbolic emulation session contained entirely within a single function and basic block, effectively capturing a concise slice of a selective symbolic emulation session. While the memmove session is bounded within one function, other symbolic sessions may span multiple functions. For instance, consider a more complex, cross-function slice that begins in Mod1.Func_A, continues into Mod2.Func_B, and then performs nested operations in Mod2.Func_C before returning to Mod2.Func_B:

```
                                                    Triggers
      Symbolic MEM load | Mod1.Func_A ldr w8, [x1] <----- Emulation
# [LogMemSymbolicDef] MEM Symdef id: f4a70000c1fe
((((SymVar_46) << 8 | SymVar_47) << 8 | SymVar_48) << 8 | SymVar_49)
      Symbolic w9       | Mod1.Func_A rev w9, w8
      Symbolic x0       | Mod1.Func_A mov x0, x9
      Symbolic RET x0   | Mod1.Func_A ret
# [LogRetSymbolicDef] RET Symdef id: c0ffee0012c4
((((SymVar_49) << 8 | SymVar_48) << 8 | SymVar_47) << 8 | SymVar_46)
      Symbolic x22      | Mod2.Func_B mov x22, x0
      Not Symbolic      | Mod2.Func_B b #0x18c328000
      Symbolic Branch   | Mod2.Func_B cbz w22, #0x18c328120
      Symbolic x0       | Mod2.Func_B add x0, x22, #8
      Symbolic Call     | Mod2.Func_B bl #0x18c2b000
# [LogArgSymbolicDef] ARG Symdef id: 1360c0000ec814
((((((SymVar_49) << 8 | SymVar_48) << 8 | SymVar_47) << 8 | SymVar_46)
 + 0x8) & 0xffffffff)
      Symbolic MEM load | Mod2.Func_C ldp x8, x9, [x0, #0x10]
# [LogMemSymbolicDef] MEM Symdef id: ba5eba11dc0c
      Symbolic x10      | Mod2.Func_C add x10, x9, #3
      Symbolic cmp      | Mod2.Func_C cmp x10, x8
      Symbolic Branch   | Mod2.Func_C b.hs #0x18c2b100
      Not Symbolic      | Mod2.Func_C ldr x8, [x0, #8]
      Symbolic MEM load | Mod2.Func_C ldr w8, [x8, x9]
# [LogMemSymbolicDef] MEM Symdef id: 3a6e00ddeca8
      Not Symbolic      | Mod2.Func_C ldrb w10, [x0, #0x44]
      Symbolic w11      | Mod2.Func_C rev w11, w8
      Not Symbolic      | Mod2.Func_C cmp w10, #0
      Symbolic w8       | Mod2.Func_C csel w8, w8, w11, ne
      Symbolic x9       | Mod2.Func_C add x9, x9, #4
      Symbolic MEM Store| Mod2.Func_C str x9, [x0, #0x18]
      Symbolic x0       | Mod2.Func_C mov x0, x8
      Symbolic RET x0   | Mod2.Func_C ret
# [LogRetSymbolicDef] RET Symdef id: 87adf00d13d4
      Not Symbolic      | Mod2.Func_B ldr x8, [sp, #0x58]
      Not Symbolic      | Mod2.Func_B sub x23, x8, #8
      Symbolic w20      | Mod2.Func_B adds w20, w0, #8
      Symbolic Branch   | Mod2.Func_B b.hs #0x18c329000
      ... Emulation continues until no more symbolic registers
```

In this multi-function session, emulation begins when Mod1.Func_A executes the guarded load at 'ldr w8, [x1]', producing a symbolic definition (MEM Symdef id: f4a70000c1fe). Emulation continues through symbolic rev, mov, and ret instructions in Mod1.Func_A, then picks up

in Mod2.Func_B with the incoming symbolic return value x0: producing a return-boundary definition (RET Symdef id: c0ffee0012c4). It branches conditionally, calls into Mod2.Func_C: producing an argument-boundary definition (ARG Symdef id: 1360c0000ec814) for the call, and then in Mod2.Func_C multiple memory loads and arithmetic operations extend the slice before returning back to Mod2.Func_B with a final symbolic return value (RET Symdef id: 87adf00d13d4) for post-call handling.

Throughout this extended session, E0 logs symbolic definitions (Symdefs) at memory loads, function returns, and argument passages to capture the precise data-flow def-use chains across-function boundaries. Definitions are captured exactly at the function-boundary events "ARG, MEM, and RET" where their symbolic expression holds the full def-use data-flow information accumulated from all previously emulated and fused symbolic slices/ sessions, tracing back to the root symbolic inputs.

The next Abstract syntax trees (ASTs) samples "collected via Triton - in AST Python Representation format" treat each SymVar_n as a root symbolic input corresponding to the n-th byte of a guarded input buffer, where E0 has already marked every byte of that buffer as symbolic. When a load reads four consecutive bytes from memory, Triton simply retrieves those preexisting SymVar_ nodes and concatenates them via nested shift/or operations to form the AST. For instance:

```
| > ldr w8, [x1]
| > loads 4 bytes from memory, resulting in w8 AST:
|   ((((SymVar_46) << 8 | SymVar_47) << 8 | SymVar_48) << 8 | SymVar_49)
|-----------------------------------------------------------------------
| > rev w9, w8
| > produces w9 AST:
|   ((((SymVar_49) << 8 | SymVar_48) << 8 | SymVar_47) << 8 | SymVar_46)
|-----------------------------------------------------------------------
| > add x0, x22, #8
| > results in x0 AST:
|   (((((SymVar_49) << 8 | SymVar_48) << 8 | SymVar_47) << 8 | SymVar_46)
|    + 0x8) & 0xffffffff
```

Emulation only ends once all live symbolic registers are concretized, at which point the system reverts to native execution with "only" guarded symbolic memory to catch the next guarded memory access and initiate the next symbolic emulation session. Where multiple symbolic emulation sessions separated by native execution gaps will all contribute in building the input data-flow def-use chains.

As another illustration, consider how a later session incorporates the full upstream def-use chain. During this run, 379 symbolic emulation sessions occurred (with native execution in between), and by session #371, a single guarded load reflects the accumulated expressions along its def-use chain from the previous 370 sessions. For example:

```
> ldr x8, [x0]
-----------------------------------------------------------------------
This instruction loads from a guarded address, producing an AST that
concatenates bytes defined (and shifted) across multiple earlier
operations in multiple earlier emulation sessions. The resulting AST is:
-----------------------------------------------------------------------
(((((((SymVar_30) << 8 | SymVar_29) << 8 | SymVar_28) << 8 | SymVar_27)
<< 8 | SymVar_26) << 8 | SymVar_25) << 8 | SymVar_24) << 8 |
  (((((SymVar_23 << (0x38 & 0x3f)) & 0xffffffffffffffff) |
      (((SymVar_22 << (0x30 & 0x3f)) & 0xffffffffffffffff) |
        (((SymVar_21 << (0x28 & 0x3f)) & 0xffffffffffffffff) |
          (((SymVar_20 << (0x20 & 0x3f)) & 0xffffffffffffffff) |
            (((SymVar_19 << (0x18 & 0x3f)) & 0xffffffffffffffff) |
              (((SymVar_18 << (0x10 & 0x3f)) & 0xffffffffffffffff) |
                (((SymVar_17 << (0x8 & 0x3f)) & 0xffffffffffffffff) |
                  SymVar_16)))))) >> 56) & 0xff))
```

This mechanism enables efficient, guided symbolic sessions fused seamlessly with native execution by using memory-access instrumentation as both the trigger and steering signal for symbolic analysis, bounding each slice to the scope of live symbolic dependencies. Unlike Concolic- execution, Selective Symbolic Instrumentation (SSI) avoids needless emulation of code unaffected by external inputs and preserves near-native performance across uninstrumented paths.

link to Phrack website for full version