



Guiding Symbolic Execution with Static Analysis and LLMs for Vulnerability Discovery

Md Shafiuzzaman, Achintya Desai, Wenbo Guo, Tefvik Bultan

University of California, Santa Barbara

KLEE Workshop 2026

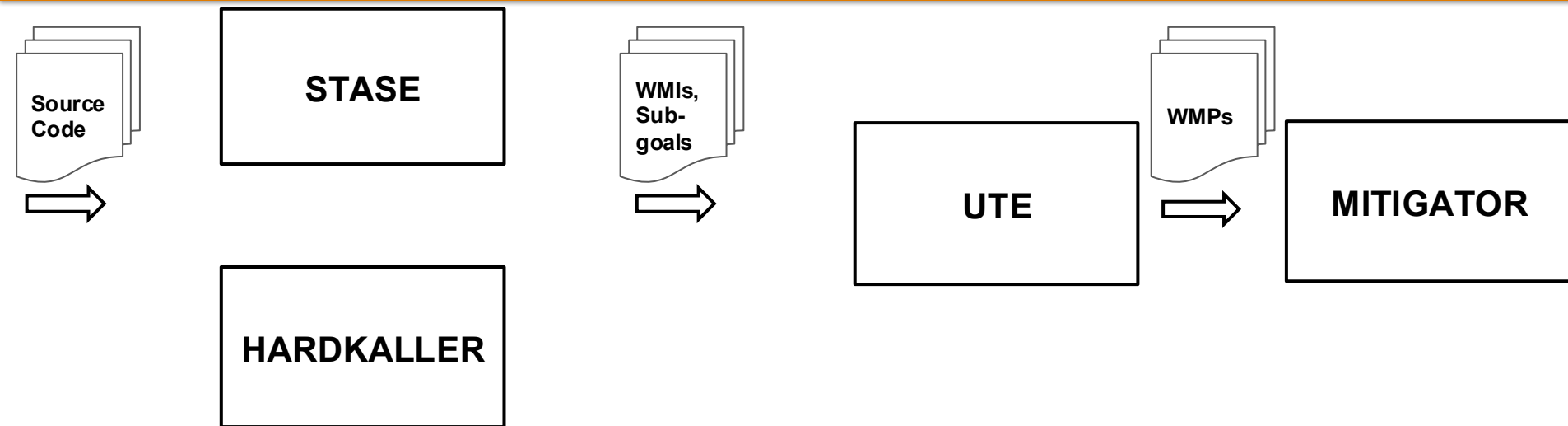
PRELUDE: DARPA HARDEN PROGRAM

HARDEN: Hardening Development Toolchains Against Emergent Execution Engines

Objective: Finding, modeling, and mitigating weird machines caused by divergences between the intended and actual behaviors of computer systems.

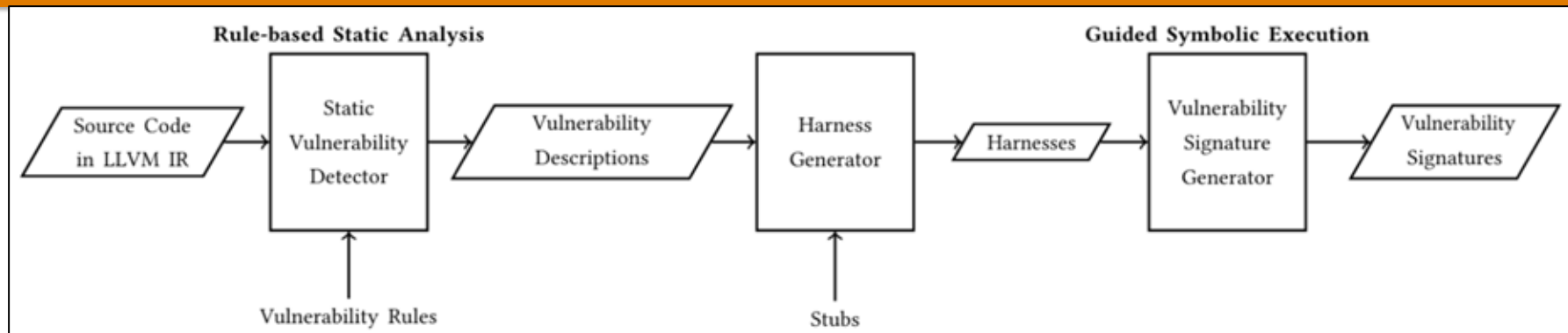
- **Weird Machine Instructions:** Vulnerabilities that can lead to unintended behaviors
- **Weird Machine Programs:** Composition of vulnerabilities that can lead to exploits

PRELUDE: UCSB HARDEN Framework + ASU UTE



- Use Symbolic Execution, Fuzzing and Static Analysis to discover primitives
- Use AI planning techniques to discover exploit chains
- Generate concrete execution traces and exploit code corresponding to exploit chains
- Use code annotations to mitigate the identified exploit chains

PRELUDE: UCSB STASE (ISSTA 2024)



Novelty:

- Combines rule-based static analysis with symbolic execution
- Generates pre- and post-conditions for discovered weird machine instructions

Key insights:

- Customizable for new weird machine instruction modeling and detection
- Static analysis guidance for generation of symbolic execution harnesses

PRELUDE: HARDEN PROGRAM, LESSONS LEARNED

- Manual effort is necessary for
 - Harnesses (entry points, inputs, environment), properties (sanitizers, assertions)
 - PoC generation
- Manual effort hurts scalability

Vulnerability Detection Techniques

	Scalability	Precision	Automation	Guarantee
Static Analysis (SA)	Yes	Prone to FP	Yes	Yes
Fuzzing	Requires Guidance	Yes	Requires Harnesses	Yes
Symbolic Execution (SE)	Requires Guidance	Yes	Requires Harnesses	Yes
LLMs	Yes	Prone to hallucinations	Yes	No

SAILOR: Combining Strengths & Removing Weaknesses

Three questions for vulnerability analysis, each handled by the technique best suited to answer it

Where does it reside?

By: Static analysis

Strength: Scales to millions of LoC with reusable rules

How is it reached?

By: LLM agent

Strength: Reasons over code to synthesize drivers and stubs

Is it triggerable?

By: Symbolic execution

Strength: Path-precise; yields a witness and a precondition

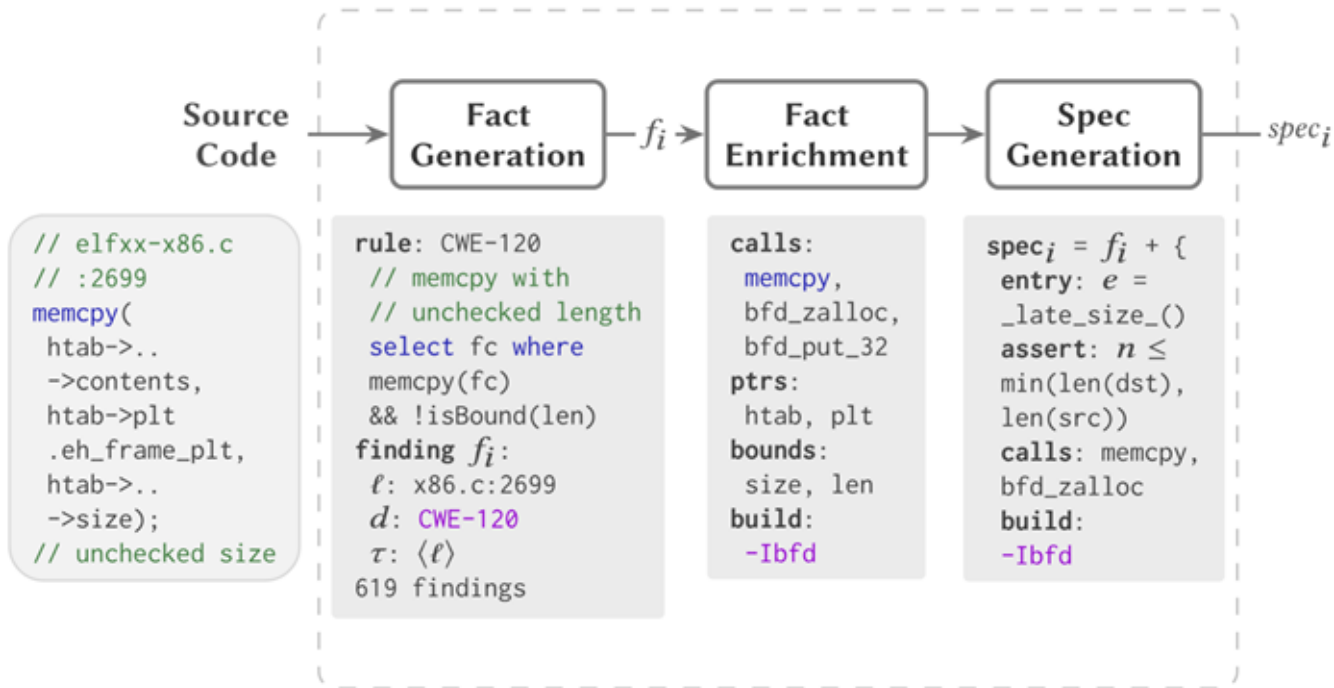
Every answer is independently validated.

Concrete replay on the unmodified source, compiled with a sanitizer, validates all three answers.

Verdicts come from KLEE and the sanitizer, **never from the LLM.**

SAILOR: Three-Phase Pipeline

Phase 1: Static analysis informed target generation



Static Analysis Rules: Written Once, Reused Everywhere

A rule pairs a memory-safety condition with the code patterns (written in CodeQL) that violate it.

45 reusable queries cover 32 memory-safety CWE classes.

Spatial (local AST): OOB write

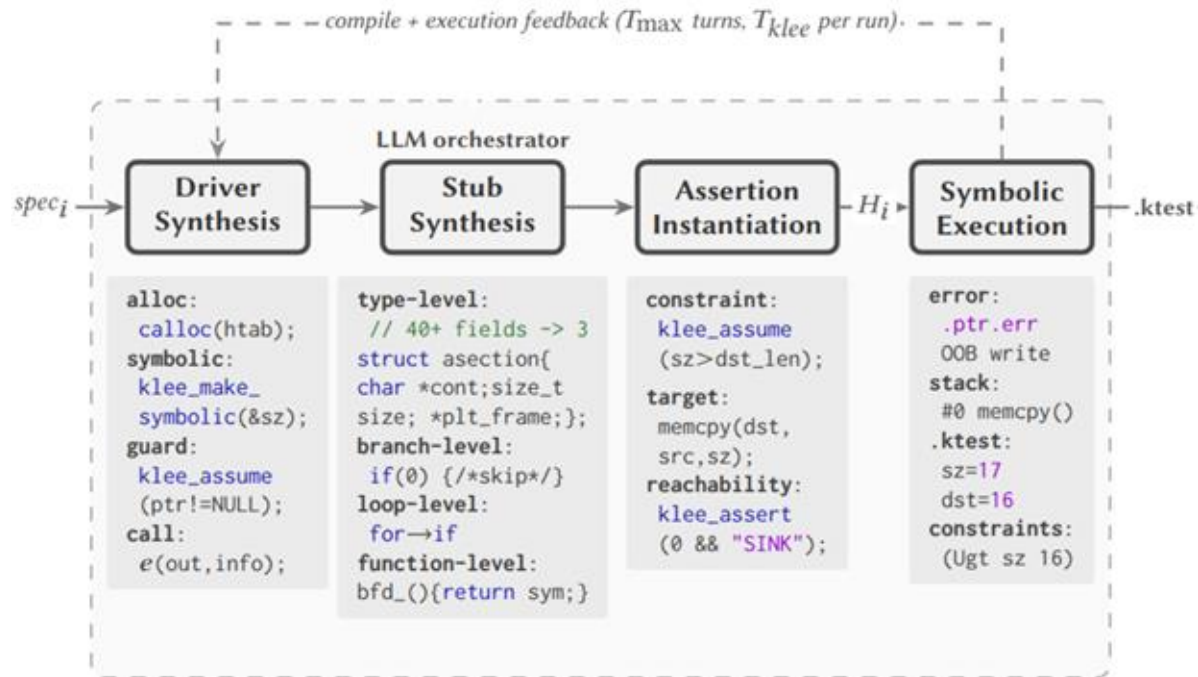
```
from Assignment a, ArrayExpr lhs
where lhs = a.getLValue()
    and interestingBase(lhs.getArrayBase())
    and not
isConstantIndex(lhs.getArrayOffset())
    and not isInsideBoundedForLoop(a, ..)
select a, "CWE-787: OOB array write to "
    + lhs.getArrayBase() + "[idx]"
```

Temporal (dataflow): use-after-free

```
from FunctionCall free, Expr use
where free.getTarget().hasName("free")
    and isDeref(use)
    and UseAfterFree::flow(
        exprNode(free.getArgument(0)),
        exprNode(use)) // no reassignment
select use, "CWE-416: use of "
    + free.getArgument(0) + " after free"
```

SAILOR: Three-Phase Pipeline

Phase 2: LLM-orchestrated symbolic execution



Environment Model: Bounding Paths Without Losing the Bug

Limit path explosion without introducing spurious paths or pruning bug triggering paths

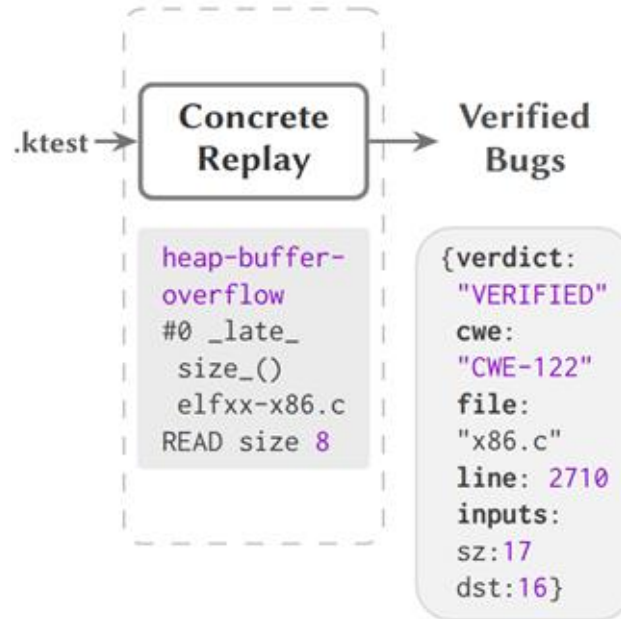
Environment model $M = \langle \sigma, W, \Sigma \rangle$

- σ , concrete setup: globals and struct fields at valid defaults
- W , symbolic input: minimal untrusted bytes; the rest concrete
- Σ , stubs: state-preserving

```
// decompress() is too costly for KLEE to execute, so it is stubbed
// X simple stub: return OK; leaves out->len = 0, so the bug is unreachable
// ✓ state-preserving stub: out->len stays symbolic; KLEE can still choose an
// overflow
int decompress(stream *s, region *out) {
    klee_make_symbolic(&out->len, sizeof(out->len), "len");
    int r; klee_make_symbolic(&r, sizeof(r), "ok");
    klee_assume(r == OK); return r; // bound the path to the success case
}
```

SAILOR: Three-Phase Pipeline

Phase 3: Concrete Validation



Concrete Validation: Confirming Witnesses by Sanitizer Replay

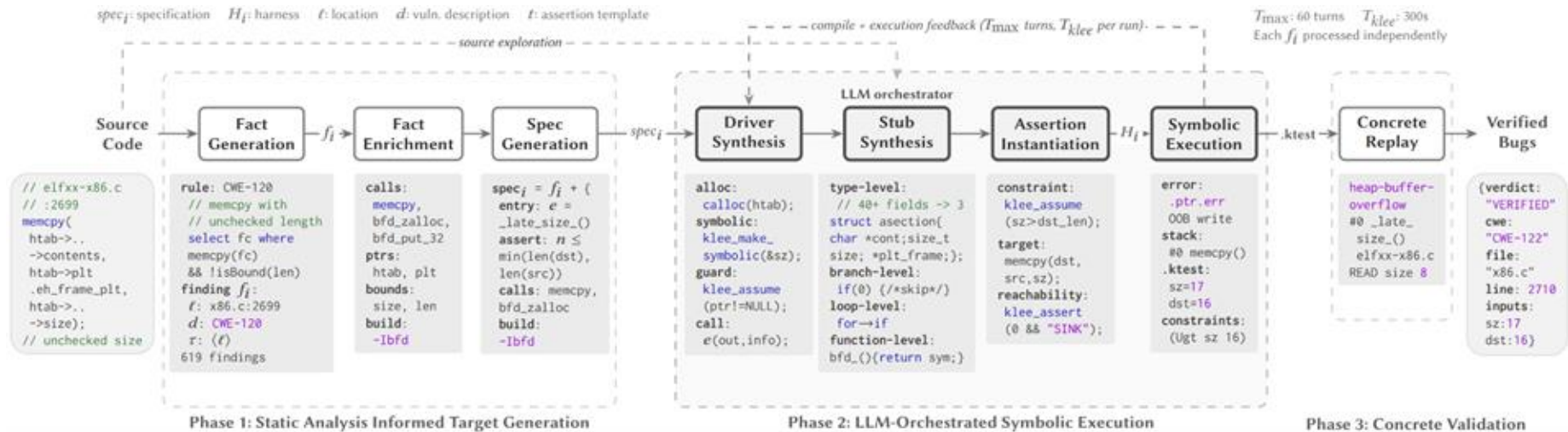
Concrete replay eliminates false positives caused by harness artifacts.

Replay the witness without the harness assumptions

- Concretize: witness bytes replace symbolic inputs; stubs dropped
- Build: the unmodified source, compiled with a sanitizer
- Confirm: the sanitizer triggers in project code, reached via the entry

```
// X rejected: the driver fabricates caller state, so the crash is a harness artifact  
char buf[16]; parse_header(buf, 64);           // declared size > allocation  
// ✓ confirmed: witness "T0" embedded in the policy, unmodified libsepol, ASan build  
read_source_policy(&parse, POL, len); link_modules(...); expand_module(...);  
==ERROR: AddressSanitizer: stack-buffer-overflow WRITE of size 8  
#0 in discard_tunables libsepol/src/expand.c:3058
```

SAILOR: Three-Phase Pipeline



Each phase strengthens the evidence: from a candidate pattern, to a KLEE witness, to a sanitizer-confirmed crash.

Example: An Out-of-Bounds Write in SELinux

libsepol: discard_tunables copies tunable terms into a fixed 10-slot array.

Phase 1 · Candidate

Sink: tmp[tunables++],
expand.c:3058

Class: CWE-787 OOB write

Entry: read_source_policy

Checker: tunables < 10

Phase 2 · Witness

KLEE: out-of-bound pointer at
expand.c:3058

Witness: term11_id = "T0" (2
bytes)

Precondition: tunables ≥ 11

Phase 3 · Confirmed

ASan: stack-buffer-overflow
WRITE at expand.c:3058

Verdict: CONFIRMED, CWE-787

PoC: through
read_source_policy

From a candidate pattern to a confirmed zero-day.

OSS-Fuzz fuzzed this library for 5 years without reaching this overflow.

Results: 10 Projects, 6.8M LoC

379
distinct
vulnerabilities

421
confirmed crashes
via concrete replay

30×
more confirmed bugs
than
the best baseline

Best baseline: agentic detection with Claude Code (full codebase access, unlimited interaction) confirmed only 12.

Reproducible: every finding is accompanied by path constraints, a concrete crash input, and a sanitizer report on the unmodified source.

SAILOR vs. LLM Vulnerability Detectors

Project	LOC	GPT-5 (conf./flag.)	SA + GPT-5 (conf./flag.)	Claude Code (conf./flag.)	SAILOR (unique / detected)
libpng	63K	0/162	0/2	0/29	21 / 29
libtiff	95K	5/72	0/64	0/7	14 / 21
libxml2	149K	0/345	0/49	0/5	11 / 16
curl	174K	0/138	0/28	0/63	0 / 0
SELinux	190K	0/30	2/81	6/28	62 / 62
OpenSSL	710K	0/818	0/194	5/157	0 / 0
SQLite	1.05M	0/99	0/10	1/43	0 / 0
mupdf	1.25M	0/0	0/219	0/37	141 / 141
FFmpeg	1.3M	0/57	0/680	0/0	78 / 78
binutils	1.84M	0/60	0/36	0/63	52 / 74
Total	6.8M	5/1781	2/1363	12/430	379 / 421

LLM-only detectors flag hundreds of issues but almost none replay on the real binary.
SAILOR has zeros on curl, OpenSSL, and SQLite, where multi-step API initialization was too hard within the harness budget, which motivates the next part.

From Internal Entrypoints to the Public API

- Question comes from maintainers: **Whether an attacker can reach the crash from outside.**
- A harness that targets internal state proves a crash, not a vulnerability.
- So in the follow-up work we require every trigger to go through a public API.

Entrypoint-level harness

```
// start at the vulnerable function
int main() {
    policydb_t *pol = calloc(1,
sizeof(*pol));
    klee_make_symbolic(pol, sizeof(*pol),
"pol");
    discard_tunables(sh, pol); // OOB write
    // Is this state reachable through
    // any public API? Not established.
}
```

API-level harness

```
// SA recovers the chain, LLM builds a
valid policy
int main() {
    read_source_policy(&parse, POL, len); //
public API
    link_modules(NULL, &parse, NULL, 0, 0);
    expand_module(NULL, &parse, &kern, 0, 0);
    // -> discard_tunables: tmp[tunables++]
    // 11 tunable terms overflow tmp[10]
}
```

Extending SAILOR to the Public API

Phase	Added for API-level vulnerabilities
Static analysis	Call-chain recovery: public API, call chain, and untrusted input bytes
LLM agent	Driver enters through public API with a concrete, format-valid input
SE and replay	Replay through public API, and the crash must depend on the symbolic input bytes

Output: An API-level PoC and a precondition for each vulnerability.

Results: 67 Zero-Days in 28 Projects (12.3M LoC)

67

confirmed zero-days
with API-level PoCs

39

in 18 AI/ML and
post-quantum projects

28

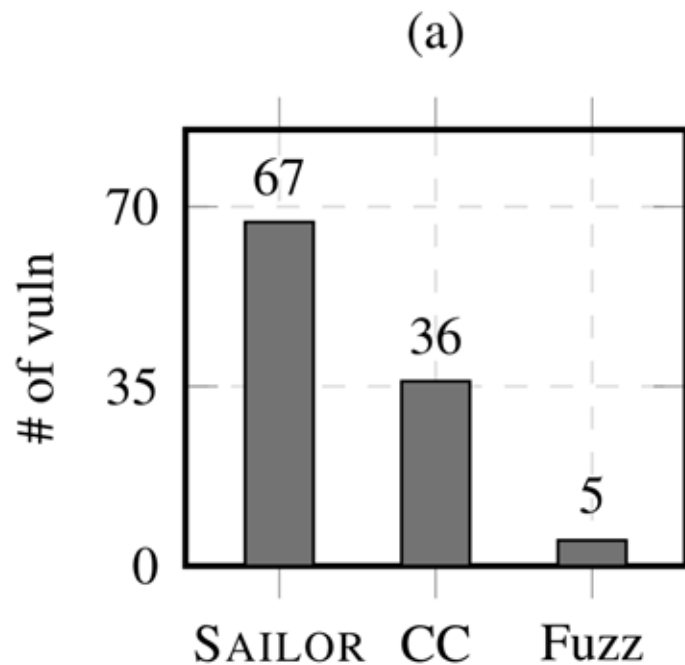
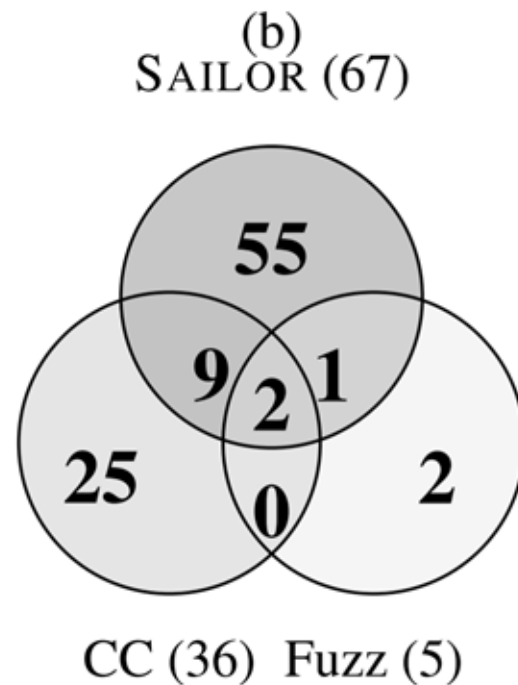
in 10 projects fuzzed
for 5–9 years

Maintainers: 14 acknowledged (10 patched, 4 in progress), 16 more were independently fixed upstream, others are awaiting responses.

Baselines: Claude Code (Opus 4.8) 36, libFuzzer 5. No false positives.

Model: SAILOR synthesizes harnesses with the less capable Sonnet 5.

SAILOR Finds 67 Vulnerabilities, 55 Are Missed by Both Baselines



(a) Vulnerabilities found, (b) Overlap among SAILOR, Claude Code (CC), and libfuzzer (Fuzz)

Selected Zero-Days

Project	Domain	Vulnerability	Status
liboqs	Post-quantum cryptography library	3 OOB read and 1 OOB write	All 4 acknowledged and patched
whisper.cpp	Speech-to-text inference	1 OOB read and 2 stack overflow	All 3 acknowledged and patched
SELinux	Linux security-policy toolchain	Double free	Acknowledged and patched
mruby	Embedded Ruby interpreter	Integer overflow to OOB write	Acknowledged and patched
koboldcpp	Local LLM inference engine	2 Stack buffer overflow	Both acknowledged and patched
stable-diffusion	Image-generation inference	OOB Read	Acknowledged and embargoed
OpenFHE	Homomorphic encryption library	5 OOB read	Reported, awaiting response

Takeaways

- **Symbolic execution scales when its harness is synthesized, not hand-written.**
 - For each vulnerability: static analysis finds where, the LLM works out how, and KLEE decides whether.
- **The LLM never decides whether a bug is real.**
 - Verdicts come from KLEE witnesses and sanitizer replay; harness errors cost recall, never precision.
- **Bounded environments, not stronger models, make the difference.**
 - Minimal symbolic input and state-preserving stubs keep the bug path feasible.
 - SAILOR outperforms Claude Code on Opus while synthesizing harnesses with Sonnet.
- **Results**
 - 379 vulnerabilities in 10 projects (6.8M LoC) from internal entry points
 - 67 zero-days in 28 projects (12.3M LoC), each with an API-level PoC and a precondition

CODA

Opinions vs. Facts: Newspaper metaphor

Newspapers have two types of writings:

- **News articles** are about **facts**:
 - They are about reporting what happened, do not editorialize, report only facts, require extensive fact checking
- **Opinion columns** are about **opinions**:
 - They are about opinions, provide personal views of the author

Opinion: *A view, judgment, or appraisal formed in the mind about a particular matter*

Fact: *Information or a piece of information presented as true or accurate*

Opinions vs. Facts

- Program analysis, symbolic reasoning, formal methods
 - These techniques generate ***facts***
 - They provide guarantees
 - ***But, scalability is a problem***
- LLMs
 - LLMs generate ***opinions***
 - Can provide opinions about anything
 - ***But, do NOT provide guarantees***

Opinions vs. Facts

Our goal is to develop trustworthy computing systems

To achieve this goal, we need to combine opinions with facts

- There are many facts we can establish, many of them are useless
- Many opinions are wrong

However,

- Opinions are critical for guiding fact generation to achieve an objective
- Fact-checking is critical for developing well-informed opinions

Opinions vs. Facts

Opinions & Facts can help each other:

- Opinions can be fact-checked
 - resulting in more trustworthy opinions
- Facts can drive opinions
 - resulting in more informed opinions

Opinions vs. Facts

Interaction of opinions and facts in SAILOR & PAGENT

- LLM generates opinions (harnesses, PoCs)
 - We use static & dynamic analysis to fact-check those opinions
- Static analysis generates facts: potential vulnerable locations
 - LLMs use those facts to generate opinions (harnesses, PoCs)

Types of Human Intelligence

- According to Nobel laureate Daniel Kahneman human intelligence has two separate components:
 - System 1: fast, instinctive and emotional
You use System 1
when you answer the question $2+2=?$
 - System 2: slower, more deliberative and more logical
You use System 2
when you answer the question $17*24=?$

Types of Human Intelligence

- Artificial intelligence has also two types
 - Type 1: Techniques based on machine learning
 - Type 2: Techniques that are based on symbolic reasoning
- Future of computing will heavily depend on **both** types of artificial intelligence
- Type 2 techniques are especially necessary for providing guarantees

Types of Human Intelligence

- We will need both
 - Type 1 Artificial Intelligence (Machine Learning) and
 - Type 2 Artificial Intelligence (Symbolic Reasoning)for achieving software dependability