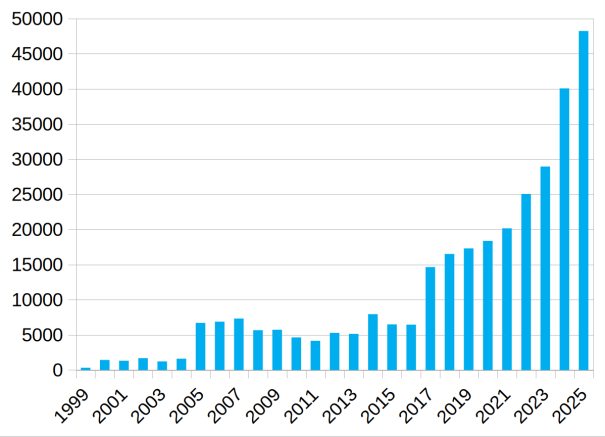
Towards Precise Vulnerability Reachability for Java

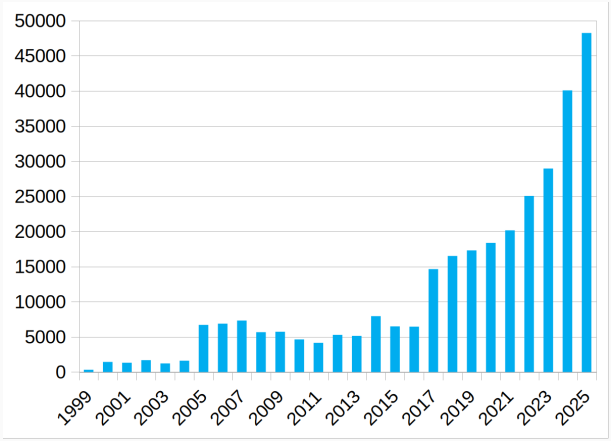
Robert B. Rubbens
Formal Methods & Tools | Semantics, Cybersecurity & Services | University of Twente

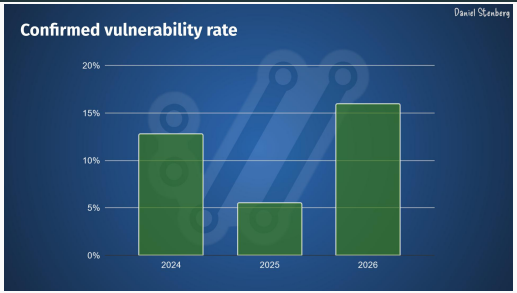
2026-07-01

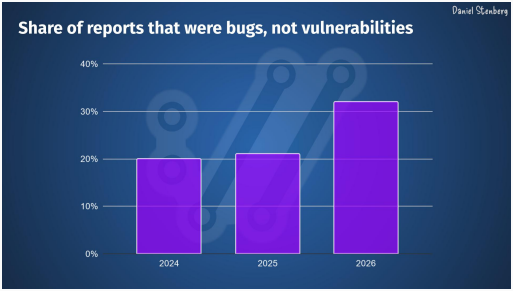
Vulnpocalypse



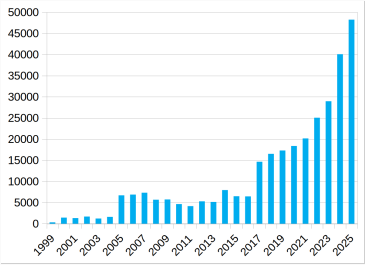
Vulnpocalypse

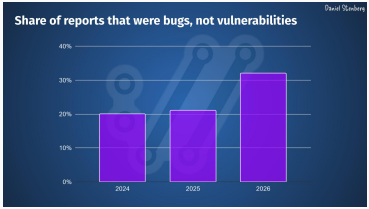






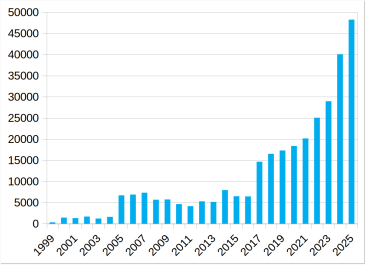
Vulnpocalypse

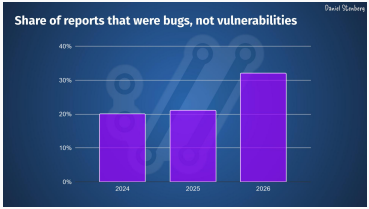




[1] endorlabs.com/learn/claude-fable-5-mythos-grade-hype

Vulnpocalypse

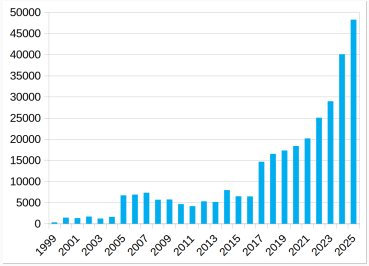


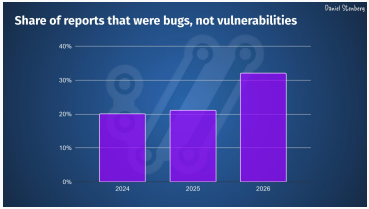


- 
- “Breakthrough”: PoC generation[1]

[1] endorlabs.com/learn/claude-fable-5-mythos-grade-hype

Vulnpocalypse





- 
- “Breakthrough”: PoC generation[1]
- Still key: quick and precise evaluation of *known* vulnerabilities

[1] `endorlabs.com/learn/claude-fable-5-mythos-grade-hype`

Overview

Q: How to quickly and precisely detect *known* vulnerabilities applications?

Overview

Q: How to quickly and precisely detect *known* vulnerabilities applications?

A: With a dash of static analysis and formal methods.

Overview

Q: How to quickly and precisely detect *known* vulnerabilities applications?

A: With a dash of static analysis and formal methods.

- Defining vulnerability reachability
- The standard solution: dependency-based scanners
- Other approaches: LLMs, trace-based solutions, bounded model checking
- Scaling bounded model checking
  - Takeaway: Java provides structure that can accelerate bounded model checking
- Future directions

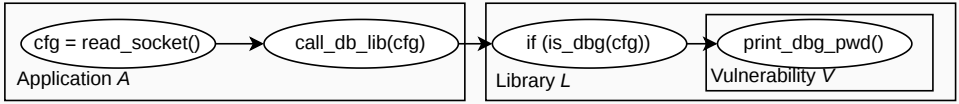
Defining vulnerability reachability

Vulnerability reachability

- Fix a library or application A, "downstream"
- Pick one of its (transitive) dependencies L, "upstream"
- A CVE V for L marks a set of vulnerable functions $f_0, \ldots, f_n$ in L
- V *affects* A when there exists a runtime trace where some f_i is reached

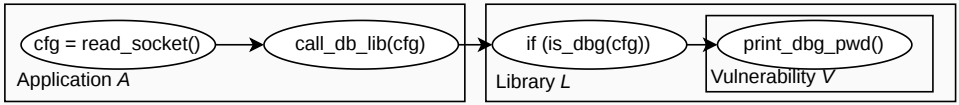
Vulnerability reachability

- Fix a library or application A, "downstream"
- Pick one of its (transitive) dependencies L, "upstream"
- A CVE V for L marks a set of vulnerable functions $f_0, \ldots, f_n$ in L
- *V affects A* when there exists a runtime trace where some f_i is reached



Vulnerability reachability

- Fix a library or application A, "downstream"
- Pick one of its (transitive) dependencies L, "upstream"
- A CVE V for L marks a set of vulnerable functions $f_0, \ldots, f_n$ in L
- *V affects A* when there exists a runtime trace where some f_i is reached



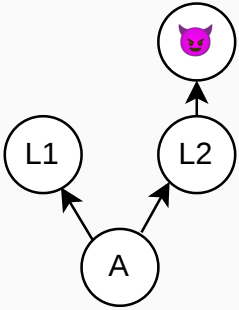
Assumptions:

- Vulnerabilities, vulnerable functions, given
- Binaries available of A, L, and any dependencies inbetween.

Dependency-based scanners

Dependency-based scanners

- Scan dependency graph for vulnerabilities
  - *V affects L*, *A depends on L*, *V affects A*
- Tools: Snyk, OWASP-DC
- Dependency check triggers: what to do?



Fixing dependency check alerts

One of the following:

- Upgrade to a fixed version $\Rightarrow$ alert goes away!
- Manually inspect, V does not affect you $\Rightarrow$ click “ignore alert”
- V is dangerous and you implement a fix $\Rightarrow$ click “ignore alert”

Fixing dependency check alerts

One of the following:

- Upgrade to a fixed version $\Rightarrow$ alert goes away!
- Manually inspect, V does not affect you $\Rightarrow$ click "ignore alert"
- V is dangerous and you implement a fix $\Rightarrow$ click "ignore alert"

Actually, sub-optimal experience for developers:

- Upgrading not always possible
  - Maintainers no longer respond to emails, upgrading breaks A, source not available
- False positive rate
- Rechecking ignored alerts

Key alert requirements

1. Be “worth it”
2. Adapt to changes over time

Other solutions

LLMs

- "Oh genie, I should be worried about CVE-2021-44228[2]? Make no mistakes."
- Principle works
  - E.g. "A Vulnerability Propagation Impact Analysis Approach Based on Code Semantics with LLM" (2024)
- General effectiveness unclear (to me). Still:
  - Cost
  - Dependence
  - Neither sound nor complete ⇒ only safe for prioritization

[2] wikipedia.org/wiki/Log4Shell

Trace-based analysis

- Idea[3]:
  - Harvest "representative" run-time traces from your application A
  - If any of the functions in the trace are in V, report
  - Otherwise, lower priority of V
- "Representative"?
  - Test suite - limited, not always available
  - Run-time monitoring of deployed system - runtime cost
- Effective!
- Still: *complementary* to dependency analysis

[3] Plate et al. (2015, 2020)

Bounded Model Checking (BMC)

BMC

- BMC 101
- Reachability with BMC
- Three optimizations:
  - Unsound method abstraction
  - Sound method abstraction
  - Faster sound method abstraction in Java

BMC 101

- Originally for verifying hardware designs
  - SAT, propositional logic
- Adapted to general software verification
  - SMT, richer logics, e.g. first-class functions
- Various tools: CBMC, JBMC, ESBMC
- Key idea: unfold program up to depth k
- Check:
  - Language properties (null pointers, division by zero)
  - Custom properties: `assert x > 0`
  - Avoiding specific locations or states: `assert false`

How to BMC

- Traverse a program, collect constraints along the way
- Bounded unrolling of looping, recursion, up to k
- Constraints satisfiable $\Rightarrow$ end of trace is reachable
- Repeat for all possible traces

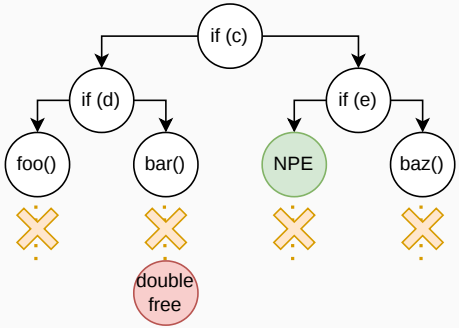
BMC in steps

```
1  if (c) {
2    if (d) {
3      foo();
4    } else {
5      bar();
6    }
7    if (e) {
8       int x = *((int*) NULL); // NPE
9    } else {
10     baz();
11   }
12 }
13
14 void bar() {
15     int* x = malloc(sizeof(int));
16     // ...
17     free(x);
18     // ...
19     free(x); // double-free
20 }
```

$k = 0$

$k = 1$

$k = 2$

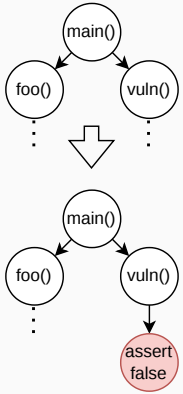


So what?

- *Very* good results:
  - Filter 95% of irrelevant bugs in dependencies (Ma et al., 2020)
  - Reduce PoC construction from weeks to ∼15 minutes (Cook et al., 2020)
- Downsides:
  - Scalability: path explosion, infinite unrollings, constraint complexity
  - Typically requires custom extensions or harnesses

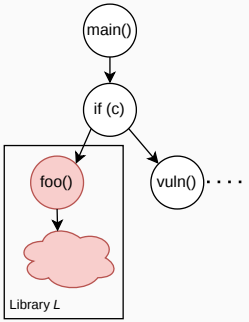
Reachability with BMC & preprocessing

1. Take the code of application and its libraries
2. Replace vulnerable function with `assert false`
3. Run the BMC:
   - Assert found ⇒ `vuln()` reachable
   - Otherwise `vuln()` not reachable



More preprocessing: dead path removal

- Many branches explored needlessly
- Remove dead paths (Ma et al. 2020)
- Performance: still not great.
- Need either small k, small systems, or harness (Cook et al. 2020)



Optimization 1: unsound method call approximation

- Insight: only unfold *some* method calls
- Abstract other calls:
  - arbitrary values written to pointers
  - arbitrary return value

Optimization 1: unsound method call approximation

- Insight: only unfold *some* method calls
- Abstract other calls:
  - arbitrary values written to pointers
  - arbitrary return value

Real execution:

```
1  int add(int* a, int b) {
2      return *a + b;
3  }
4
5  int v = 0;
6  int* p = malloc(sizeof(int));
7  *p = 3;
8  // State: v == 0, *p == 3;
9  v = add(p, 5);
10 // State: v == 8, *p == 3
```

Optimization 1: unsound method call approximation

- Insight: only unfold *some* method calls
- Abstract other calls:
  - arbitrary values written to pointers
  - arbitrary return value

Real execution:

```
1  int add(int* a, int b) {
2      return *a + b;
3  }
4
5  int v = 0;
6  int* p = malloc(sizeof(int));
7  *p = 3;
8  // State: v == 0, *p == 3;
9  v = add(p, 5);
10 // State: v == 8, *p == 3
```

BMC execution:

```
1  int add(int* a, int b) {
2      return *a + b;
3  }
4
5  int v = 0;
6  int* p = malloc(sizeof(int));
7  *p = 3;
8  // State: v == 0, *p == 3;
9  v = add(p, 5);
10 // State: v = fresh(), *p = fresh();
```

Optimization 1: inaccuracy

```
1  int* total = malloc(sizeof(int));
2  int add(int* a, int b) {
3      *total += b;
4      return *a + b;
5  }
6
7  int v = 0, total = 0;
8  int* p = malloc(sizeof(int));
9  *p = 3;
10 // State: v == 0, *p == 3, *total == 0;
11 v = add(p, 5);
12 // State: v == fresh(), *p == fresh(), *total == 0;
```

Optimization 1: inaccuracy

```
1  int* total = malloc(sizeof(int));
2  int add(int* a, int b) {
3      *total += b;
4      return *a + b;
5  }
6
7  int v = 0, total = 0;
8  int* p = malloc(sizeof(int));
9  *p = 3;
10 // State: v == 0, *p == 3, *total == 0;
11 v = add(p, 5);
12 // State: v == fresh(), *p == fresh(), *total == 0;
```

- Tradeoff: fast & scalable
- But: inaccurate (global state, pointers to pointers)

Where to apply optimization 1?

- "Deeper" function calls on trace
- Library calls not directly related to the vulnerable functions

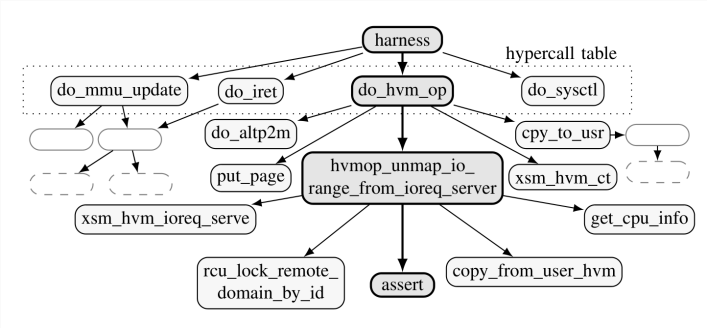


Figure 1: (Cook et al., 2020)

Optimization 2: sound method call approximation

- Again: only unfold *some* method calls, abstract others
- Also, arbitrary return value
- But: reset the *entire* heap memory to arbitrary values

Optimization 2: example

```
1  int* total = malloc(sizeof(int));
2  int add(int* a, int b) {
3      *total += b;
4      return *a + b;
5  }
6
7  int v = 0, total = 0;
8  int* p = malloc(sizeof(int));
9  *p = 3;
10 // State: v == 0, *p == 3, *total == 0;
11 v = add(p, 5);
12 // State: v == fresh(), *p == fresh(), *total == fresh();
```

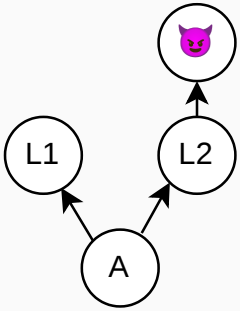
Optimization 2: example

```
1  int* total = malloc(sizeof(int));
2  int add(int* a, int b) {
3      *total += b;
4      return *a + b;
5  }
6
7  int v = 0, total = 0;
8  int* p = malloc(sizeof(int));
9  *p = 3;
10 // State: v == 0, *p == 3, *total == 0;
11 v = add(p, 5);
12 // State: v == fresh(), *p == fresh(), *total == fresh();
```

- Expectations:
  - Bad performance: suddenly possible NULL everywhere
  - But: accurate when it works

Optimization 3: Sound method call approximation in Java

- Leverage strong structure of Java's memory model
- Only reset *part* of the heap
- Intuition: L1 does not affect A, L2, etc.
- Challenges:
  - Lambdas, inheritance break the separation assumption
  - E.g. `L1.foo( () -> L2.set(3) )`
  - BMCs might need to be adapted for per-field heaps



Future directions & challenges

Infinite loop unrolling $\Rightarrow$ path explosion

Infinite loop unrolling $\Rightarrow$ path explosion

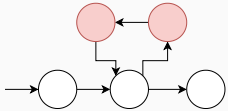


Figure 2: Constrained Horn Clause-based analysis

Infinite loop unrolling ⇒ path explosion

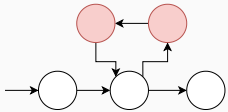


Figure 2: Constrained Horn Clause-based analysis

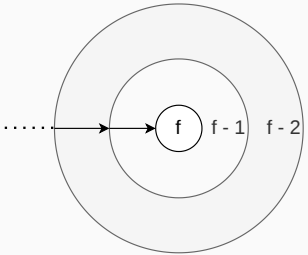
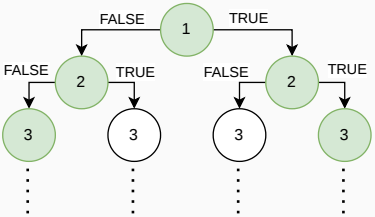


Figure 3: Generic efficient traversal schemes

“Operational model”

- What about files, sockets, windows registry, ...?
- Operation model captures system call behaviour
- Default: overapproximation
  - Safe, but wasteful
  - All interleavings of files popping in and out of existence are considered

```
1  Paths.get("/etc/passwds").exists();
2  Paths.get("/etc/passwds").exists();
3  Paths.get("/etc/passwds").exists();
```



- Prior work: Angr, JBMC
- No systematic prior work (afaik)

Conclusion

- Vulnpocalypse is still going strong
- Checking for known vulnerabilities is still open
- Good software engineering works
- But: can do better with less effort
- BMC: solid foundations for precise analysis

Near future work:

- Extend & experiment with JBMC, ESBMC-Java, JayHorn
- Measure effectiveness of different abstraction approaches