Precise Vulnerability Reachability for Java

Robert B. Rubbens, Andrea Continella (SCS), Marieke Huisman (FMT), University of Twente

Abstract

Developers are flooded with CVEs every day. Automated methods for filtering are too coarse. We propose a method that flags vulnerabilities based on *semantic* path analysis. For this, we adapt earlier work by Ma et al. on bug reachability in the data science domain. We will evaluate our approach by reusing an existing dataset of vulnerable libraries. This approach will reduce time required for CVE reports, and increase time available for meaningful work.

Introduction

Developers are flooded with CVEs every day. A CVE is a report of a vulnerability in a piece software. Their number has been increasing steadily over the years, as shown in fig. 1.

To tame the influx of CVEs, developers use automated tools for initial filtering. Unfortunately, these tools are not very effective, as the level of granularity is too coarse. This causes many CVEs to be flagged falsely, demotivating developers to follow up on vulnerability warnings. We are developing a *precise* and *reliable* vulnerability analysis method that flags vulnerabilities based on *semantic* path analysis.

To accomplish this, we adapt earlier work by Ma et al. on bug reachability in the data science domain, and generalize it to a Java application context. We essentially soundly approximate real execution by bounded unrolling of recursion and abstracting methods.

We will evaluate our approach by reusing an already existing dataset of application-CVE-library triples by Wu et al. based on call graph analysis. This allows both ground truth checking as well as benchmarking of our prototype.

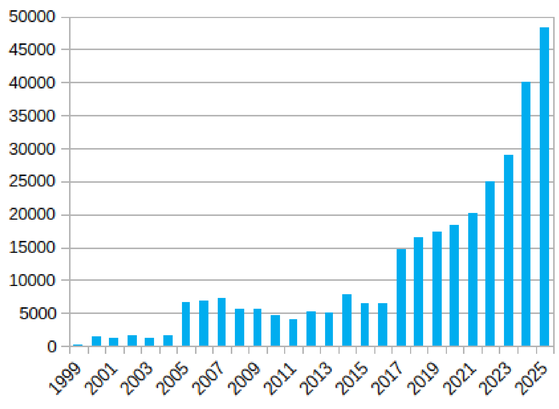

Figure 1: Number of new CVEs per year

Reachability

An application a or library l depends on n libraries $d_1, \ldots, d_n \subseteq L$. We define a set of entry points E, which is either the set of main functions or public API functions. Each application a or library $l \in L$ has m dependencies $d^l_1, \ldots, d^l_m \subseteq L$. We assume existence of a small step semantics for the programming language to characterize actual executions, written as $s_1 \to s_2$. Meaning, given a state s_1, a program can take one execution step to state s_2. We can now define three levels of reachability relations:

- **Dependency graph (DG)**: l_1 can DG-reach l_k if there exists a sequence of $l_1, \ldots, l_k$ where l_i depends on l_{i+1}.
- **Call-graph (CG)**: f_1 can CG-reach f_k if there exists a sequence $f_1, \ldots, f_k$ where each f_i has a statement s that calls f_{i+1}.
- **Semantic reachability:** s_1 can semantically reach s_k if there exists a sequence of states $s_1, \ldots, s_k$ where $s_i \to s_{i+1}$.

Bug $\iff$ vulnerability

Bug reachability can be direclty repurposed to implement *vulnerability* reachability by replacing the *buggy* right-hand side of a reachability relation with a *vulnerable* one.

The Ma et al. encoding

Ma et al. check semantic reachability by encoding programs into SMT using the following rules:

Method calls:

- Local functions: unfold
- Library functions: unfold or summarize
- Else: havoc state (overapproximate)

Loops:

- Unroll 10 times
- Beyond: havoc state

Dead ends: terminate early.

Potential reduction

Ma et al. have shown that in a Python data science context, their analysis can reduce the number of warnings about reachable bugs by **95.6%**. [2]

	DGR	CGR	SR
Bug	SonarQube	Sonarqube	Ma et al.
Vuln.	OWASP DC	Plate et al.	This work

Table 1: Non-exhaustive previous related work

Challenges

It remains to be shown if the analysis by Ma et al. scales up to a more general context. We expect at least the following challenges to reduce the effectiveness of reachability filtering as reported by Ma et al. We think we can address some of them.

Heap encoding: Ma et al. do not describe how they handle mutable heap locations. It might be the case that the data science fragment of Python is checkable without considering a mutable heap. We expect that for meaningful analysis of Java applications an accurate encoding of the heap is vital.

Class hierarchy size: path explosion because of inheritance could be a problem. In contrast, in a data science context the type hierarchies are shallow. To counteract this problem, we will include type info in the symbolic state.

Lambda handling: in Java, lambda objects are subclasses of functional interfaces, so suffer from the same problems as deep class hierarchies. As their use is more local, we expect local ad-hoc optimizations to be sufficient.

Native behaviour modeling: abstractly modeling native behaviour might introduce inaccuracies in the path condition, degrading precision of the analysis.

State space explosion: this is a well-known problem in program analysis. Once we have a functioning prototype, we will tackle this with gradual search algorithms.

Prototype

We are currently developing a prototype in Scala. The input of the tool is an application or library consisting of jar files, transitive dependencies and the Java runtime. These are imported using SootUp [4] into a "mega-CFG" and type-hierarchy. For each $f \in E$, we collect path conditions, assignments and heap mutations into an SMT formula p. When a vulnerable function f_v is reached, satisfiability of p implies reachability.

Evaluation

To evaluate effectiveness, we use a baseline dataset by Wu et al. containing triples (library, CVE, dependency) with an accompanying CG-reachability trace. We will use this to establish correctness by checking if the traces from the dataset also exist in our mega-CFG, i.e. by stripping path conditions before checking satisfiability of p. In addition, we will also use the dataset to check for feasibility as the dataset simulates realistic test cases.

Future work

We want to explore promising search strategies to combat state space explosion. For example, we are considering a middle-out search strategy that checks reachability by first considering f_v, then its callers, and so on. Beyond that, we want to consider the standard prioritization strategies: depth-first, breadth-first and least unrollings.

If this approach is successfully generalized, we will apply it to other languages.

References

[1] H. Plate et al. "Impact assessment for vulnerabilities in open-source software libraries". In: 2015. DOI: 10.1109/ICSM.2015.7332492.

[2] W. Ma et al. "Impact analysis of cross-project bugs on software ecosystems". In: 2020. DOI: 10.1145/3377811.3380442.

[3] Y. Wu et al. "Understanding the Threats of Upstream Vulnerabilities to Downstream Projects in the Maven Ecosystem". In: 2023. DOI: 10.1109/ICSE48619.2023.00095.

[4] K. Karakaya et al. "SootUp: A Redesign of the Soot Static Analysis Framework". In: 2024. DOI: 10.1007/978-3-031-57246-3_13.

Contact Information

- r.b.rubbens@utwente.nl
- bobrubbens.nl

SEMANTICS, CYBERSECURITY & SERVICES (SCS)

Formal Methods & Tools

UNIVERSITY OF TWENTE.