The Dynamics of Software Composition Analysis

Darius Foo
Veracode
dfoo@veracode.com

Jason Yeo
Veracode
jyeo@veracode.com

Xiao Hao
Veracode
haxiao@veracode.com

Asankhaya Sharma
Veracode
asharma@veracode.com

arXiv:1909.00973v2 [cs.SE] 30 Sep 2019

***Abstract*—Developers today use significant amounts of open source code, surfacing the need for ways to automatically audit and upgrade library dependencies, and giving rise to the subfield of Software Composition Analysis (SCA). SCA products are concerned with three tasks: discovering dependencies, checking the reachability of vulnerable code for false positive elimination, and automated remediation. The latter two tasks rely on call graphs of application and library code to check whether vulnerability-specific sinks identified in libraries are used by applications. However, statically-constructed call graphs introduce both false positives and false negatives on real-world projects. In this paper, we develop a novel, modular means of combining call graphs derived from both static and dynamic analysis to improve the performance of false positive elimination. Our experiments indicate significant performance improvements.**

***Index Terms*—Software analysis, Software security and trust; data privacy**

I. Motivation

Developers today use large amounts of third-party code to build applications, as code reuse significantly increases productivity and lowers development costs [25]. However, given the fact that up to 80% of typical applications are now third-party code, there is a growing need for tools to manage open source risk [27]. The 2017 Equifax data breach was infamously caused by a vulnerability in Apache Struts 2,[1] an open source web framework. "Using Components with Known Vulnerabilities" is listed in the OWASP Top 10 [23], and the dependency upgrades required to remediate these are time-consuming and often not carried out frequently enough [7] [18] [8]. Furthermore, application dependencies constantly change and are difficult to determine and audit manually due to the complexity of modern package managers and library ecosystems.

Software Composition Analysis (SCA) is an emerging subfield of application security concerned with precisely this problem. SCA products offer a suite of services centered around automated identification of third-party library dependencies. Auxiliary services such as interfaces for viewing software inventories, enforcing organization-wide policies, and integration with CI/CD [9] setups may also be present.

Our SCA product solves two pain points with typical SCA offerings. The first is false positives arising from straightforward dependency analysis – framework-based applications pull in large trees of transitive dependencies, which, despite the fact that they are included, may not be used in the final application, or at least not in vulnerable ways. We analyze application call graphs to identify and eliminate such cases. The second pain point is remediation: we provide a way to automatically upgrade dependencies, using call graphs again to check if the upgrade is potentially breaking.

In this paper, we describe the architecture of a state-of-the-art SCA product and discuss techniques for performing the core SCA tasks: detection of libraries, reachability of vulnerable code for false positive elimination, and automated remediation. We motivate the need for dynamic analysis to obtain accurate results across all tasks and illustrate a novel means of composing call graphs derived from static analysis and instrumentation in a manner that is modular in third-party libraries, allowing analysis to be performed scalably in CI/CD pipelines. Merging static and dynamic call graphs can result in significantly more vulnerable methods discovered, improving the performance of downstream tasks.

[1] https://nvd.nist.gov/vuln/detail/CVE-2017-5638

II. Related Work

The problem of false positives in static call graph construction has been thoroughly investigated [16] [29], and attempting to improve precision using dynamic analysis is a natural next step [28] [30]. Blended analysis [21] [22] is a similar approach, first obtaining the structure of a program using dynamic analysis, then applying a static analysis on top of it.

[26] is perhaps the most related work, similarly utilizing a combination of static and dynamic analysis to eliminate false positives in SCA. The key innovations of our approach are the use of hand-curated, vulnerability-specific sinks and the fact that the analysis is modular, not requiring call graph construction on demand for libraries (an idea similar to [31]). The latter makes the analysis scalable, allowing it to complete in minutes on average and making running it in a CI pipeline feasible.

III. Discovering Dependencies

An essential problem in SCA is that of *discovering dependencies*: determining the third-party libraries a project uses given its source code. Results are typically drawn from some universe of open source libraries, such as coordinates on Maven Central.

A. Static Dependency Analysis

The most straightforward way of determining a project's dependencies is to read its *dependency manifests*, e.g. `pom.xml`. These contain a listing of library coordinates and associated version constraints. Package managers interpret these manifests to perform *dependency resolution*: querying an external

repository to determine the transitive dependencies of each library (which may themselves introduce more constraints), then selecting a set of libraries which satisfies all constraints. In the event of an inability to satisfy every constraint, package managers may fail or approximate a solution (e.g. Maven's *nearest definition* heuristic, which applies when multiple versions of the same library are required). The definition of what constitutes a valid solution also varies, e.g. npm does not require a single version of each library.

B. Dynamic Dependency Analysis

The main difficulty of detecting dependencies statically is in modeling the implementation of a package manager correctly. Package managers are typically unspecified and must be each handled specially. Dependency resolution is also often nondeterministic and might produce different results over time, due to reliance on external repositories that may be updated by developers. This suggests that a better approach is to perform a dynamic analysis instead; a theme we explore in this paper.

A dynamic dependency analysis integrates with package managers and can get the results of dependency resolution exactly as they would have produced at a point in time. The main shortcoming is that a full build of the project may be required, which is time-consuming and difficult to automate in general, due to the system dependencies and configuration required.

Nevertheless, going from a purely static analysis to a package-manager-integrated dynamic analysis demonstrably improves accuracy. We evaluate this using a set of 41 microbenchmarks[2] across 18 different package managers. On average, we see that the dynamic analysis improves the number of dependencies discovered by 204% (131 with purely static, and 398 with dynamic), with no cases in which it performs worse.

IV. Identifying Vulnerabilities

A. Vulnerability Databases

The next step is to identify vulnerabilities due to libraries. A baseline data source is the NVD; practically every SCA product informs users of CVE vulnerabilities. SCA vendors also curate their own vulnerability datasets, supported by both data mining methods [3] [14] and custom tooling [2].

Determining if a vulnerability is exploitable in an application would allow an SCA product to report fewer false positives. This is valuable because notification fatigue from false positives becomes the bottleneck when fixes are deployed automatically [8]. Furthermore, the amount of false positive reduction is significant: commercial SCA products report that 70-80% of dependencies are never referenced in application code.

Our approach is based on a lightweight reachability analysis using call graphs. We discuss the tradeoffs of various call graph construction methods before covering our implementation of the analysis and how it fits into our SCA product.

[2]https://github.com/srcclr/efda

B. Static Call Graphs

Given an application's source code, we may view it as a *call graph*, a structure that expresses the calling relationships between its methods. A directed edge e from method m_1 to method m_2 indicates that a call site within m_1 invokes m_2 in some execution of the program. A *call chain* is a sequence of edges $e_1 e_2 ... e_n$ where $(m_n, m_{n+1}) = e_n$ and $(m_{n+1}, m_{n+2}) = e_{n+1}$, i.e. consecutive edges must share a method vertex.

Call graph construction for object-oriented programs must model the effects of dynamic dispatch correctly. False positives are possible when applying such analyses [15], leading to *infeasible* edges which are present in the graph but do not occur in any concrete run of the program. Furthermore, *call chains* comprising only feasible edges may themselves prove be infeasible. On average, 25.5% of call chains created by static approaches are infeasible [16].

The presence of infeasible call chains means that a reachability analysis would produce false positives, traversing paths which will never occur at runtime.

C. Dynamic Call Graphs

Real-world static analysis tools often deliberately eschew prohibitively expensive or conservative approximations that would render analysis less useful when "hard" language features such as reflection are involved [19]. This causes problems when analyzing highly dynamic frameworks, such as Spring or Rails: the resulting call graphs are incomplete and lead to false negatives in reachability analyses.

An alternative is the use of runtime instrumentation, a form of dynamic analysis. Executing a program's tests and recording its flows has the upside of never producing any infeasible call graph edges, since only flows which were actually observed are reported. Support for all language features is also a given.

The downsides are that this approach produces false negatives (missing paths to sinks called by code not covered by tests) and requires code to actually build and run in order to be analyzed. Instrumentation also imposes overhead, slowing test execution.

D. Combined Static and Dynamic Graphs

The relationship between static and dynamic call graphs, as well as how they approximate a *theoretically ideal* call graph, has been investigated by [28]. In short, the ideal call graph is the union of dynamic call graphs across all possible executions of a program (making the dynamic call graph for one execution a subset of it), and the static call graph is a superset of the ideal call graph due to the presence of infeasible edges. Due to the soundiness [19] of our approach (because of false negatives from unsupported language features; covered in detail in Section IV-E1), we modify the definition slightly, so that neither the static call graph nor the ideal call graphs are subsets of each other. The modified relationship is shown in Figure 1.

The final call graph that we search for uses of vulnerability-specific sinks is derived by taking the union of the static

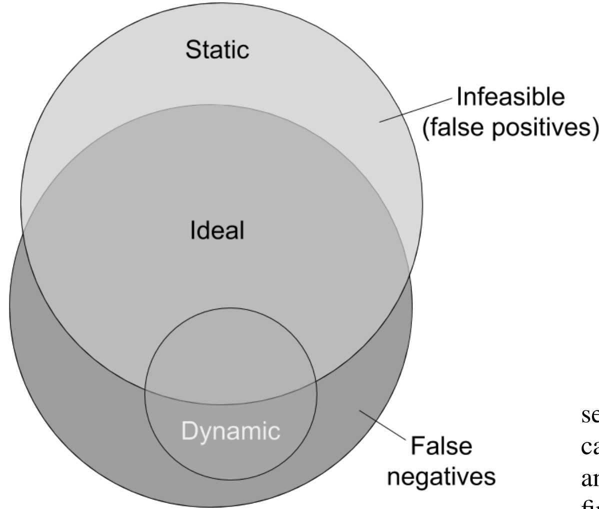

Fig. 1. Combined static and dynamic call graphs

and dynamic call graphs, the intuition being that it brings us closer to the ideal call graph. In the following sections, we explain how this is done in a modular fashion, preserving the properties of the graph that allow a fast reachability check.

E. Vulnerable Methods

We consider a vulnerable library to be *possibly used* in an application if a vulnerability-specific sink is reachable from the application's call graph. This is weaker than determining if a vulnerability is *exploitable* as control flow is not taken into account, admitting false positives. We curate these sinks by hand; these are the method-level root causes of vulnerabilities mentioned in CVEs and proprietary vulnerabilities.

1) Static Call Graph Construction: To illustrate our approach, we refer to Figure 3, which shows the final graph, with labels for vertices (e.g. A) and contours for subgraphs (e.g. CC). Given an application, we start by constructing a static call graph $G_s = (V_s, E_s)$, represented by the contour G_s. As we only analyze the application, V_s consists of both application methods (B, A, D, E) and the entry points of libraries called *directly* by the application (C, U); transitively-called library methods (V, P, Q) are absent.

E_s is initialized to the set of statically-known static or virtual calls; given a method call $a.b()$ in method m, we add an edge between m and the method b of a, using a's declared class or interface.

We expand the set E_s with a number of passes:

1) Class Hierarchy Analysis (CHA) [4], which determines possible receiver classes for b using subclass relations
2) Rapid Type Analysis (RTA) [5], which rules out receiver classes using information about object instantiations
3) Reflection analysis, which adds edges for reflective calls with constant arguments; as this is a subset of all possible edges due to reflective calls, the approach is soundy [19]

```
1: for e in e_1 e_2 ... e_n do
2:     (m_1, m_2) ← e
3:     if m_1 ∈ V_s then
4:         add e...e_n to G'_s
5:         break
6:     end if
7: end for
```

Fig. 2. Merging vulnerable method call chains

Thus we have $G'_s = (V_s, RTA(CHA(E_s)))$. We define the set of *first-party entry points* V_e as the set of methods without callers, i.e. $\{m_1 \mid \exists m_1 \in Vs, \forall (m_2, m_3) \in E_s, m_1 \neq m_3\}$. A and D in the diagram are examples. Methods of V_e must be first-party, as third-party methods must be called by a first-party method to appear at all. V_e may be further filtered down (e.g. by considering only `main` methods) depending on analysis goals. G'_s is constructively a graph of all methods reachable from a first-party entry point.

Separately, we precompute *vulnerable method call chains* CC for libraries: sequences of directed edges $e_1 e_2 ... e_n$ such that e_n ends at a vulnerable method (Z, V, Q), and e_1 is an entry point of the library (Y, X, U, P). A call chain represents a path from a library entry point to a library-specific sink.

2) Merging Library Call Chains: Given a vulnerable method call chain $e_1 e_2 ... e_n$ and a static call graph G'_s, we merge them using the algorithm in Figure 2. For each call chain, we iterate through its edges (line 1), looking for a suffix which begins at an existing edge in the graph (lines 2-3), adding the suffix to the graph if we find it (line 4).

Applied to G'_s, this would result in V being added. The approach ensures that we do not introduce third-party entry points (X, Y, P): new outgoing edges are only attached to existing ones and no new incoming edges are added. This preserves the property that all vertices are reachable from a first-party entry point. Consequently, determining if a vulnerable method is called in some execution of the program – the key question we are interested in answering – is a simple membership check.

3) Dynamic Call Graph Construction: Given tests, we instrument them to derive a dynamic call graph, then compose it with the static call graph and vulnerable method call chains to find more potentially reachable vulnerable methods.

Running the tests, we construct a dynamic call graph $G_d = (V_d, E_d)$. V_d comprises both first- and third-party methods (J, T, S, R, B), and E_d contains only feasible edges and paths. In contrast to E_s, where method callers were only ever first-party methods and callees were either first- or third-, E_d contains third-party callers *and* callees – this may be observed from the execution of a framework such as JUnit, where a `main` method defined in JUnit itself (J) is called, which dynamically discovers `@Test`-annotated user methods (T) to invoke reflectively. This *inversion of control*

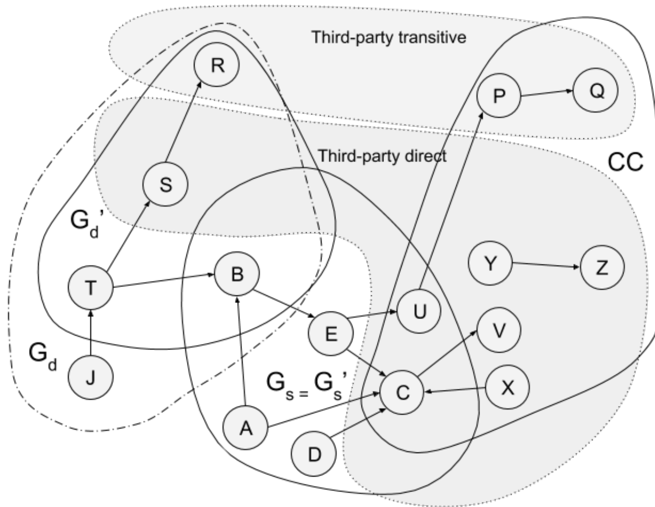

Fig. 3. Example of a composed call graph. The boundary lines represent (from left to right) the sets G_d, G'_d, $G_s = G'_s$, methods of a direct dependency, methods of a transitive dependency, and CC.

is a common pattern in framework-based applications [26]. First-party entry points (T) thus *do* have callers.

When composing the static and dynamic call graphs, we would like to preserve the property that allowed us to determine reachability using set membership. We have to add entry points now, though, because tests were never considered earlier; the solution is then to ensure that they are first-party.

We first instrument tests, and when the test run completes (either in success or failure), the resulting edges form the graph G_d. Next we identify *framework entry points*: these are the edges which span framework and application code (JT). We identify frameworks manually here as there is no good way to differentiate them without more context, special-casing common ones like JUnit and TestNG.

Finally, given a framework entry point, we take the vertices of its transitive closure (T, S, R, B) and add it to a new graph G'_d – a graph of dynamic edges whose entry points are first-party.

The union of G'_s and G'_d gives us G_c, a combined static and dynamic call graph with only first-party entry points. We may further merge the static call chains into G_c. Paths in G_c may then span both static and dynamic edges.

F. Discussion

A limitation of vulnerable method call chains being computed for single libraries in isolation is that vulnerable methods called *only within third-party code* will be missed. For example, the edge UP will not be in any call chain because we cannot tell when computing call chains that P is the start of a vulnerable method call chain from another library. The dynamic call graph does handle this case, however, as shown by S and R; if there were an edge RP (analogous to UP), we would be able to detect the vulnerable method call.

V. Remediation

Another task we perform automatically is remediate library vulnerabilities. Our approach [1] is a static analysis that precomputes diffs between library versions to determine the changes between them. The diffs are augmented with call graphs (constructed as in Section IV-E) and are thus *semantic* in nature, e.g. considering methods changed if their callees change.

We then check if methods in the diffs of potential library upgrades occur in the application's call graph. These are shown as potential breaking changes in pull requests that we create automatically, indicating to developers which upgrades are riskier.

A. Dynamic Analysis

In keeping with the theme of the paper, we outline an extension to the above analysis which uses an instrumentation-derived call graph to improve accuracy. The analysis depends on accurate call graphs when computing diffs with semantic information, and checking if an application uses a method of a combined diff. The latter may benefit from this, especially if there are tests available. Given that a dynamic call graph will not contain infeasible edges, there will be not be additional false positive calls to libraries, leading to fewer library upgrades being considered as breaking when they are not.

There is also the obvious "dynamic analysis": executing tests to check if an upgrade introduces breakage. This does not subsume the entire analysis for breaking changes as the latter reveals changes occurring outside the coverage of the test suite and non-breaking semantic changes.

VI. Evaluation

TABLE I
CALL GRAPH EDGES

project	static vertices	static edges	dynamic vertices	dynamic edges	static sinks	dynamic sinks
helios	7930	26746	12287	39813	1	3616
immutables	30206	319934	398	874	5	5
java-apns	536	999	4240	8685	58	859
retrofit	1925	5269	7339	22565	6	6

We evaluated the performance of call graph construction on four real-world Maven-based Java projects (Table I). On average, we find that dynamic call graphs add 824% more vertices and 361% more edges, allowing us to discover significantly more call sites from which vulnerable methods are reachable in 2/4 cases (shown in the **static sinks** and **dynamic sinks** columns). Most of the extra edges are from third-party dependencies. The tradeoff is that dynamic call graphs less easy to apply automatically, as manual effort is required to configure projects so that they compile successfully and their tests run at least partially. There is also significant variance between projects in test coverage (and correspondingly, dynamic call graph size and vulnerable method reachability).

VII. Conclusion and Future Work

We motivated and described the SCA problem: the fact that major portions of modern applications are third-party emphasizes the need for tooling to automatically audit and upgrade dependencies. Our approach has three components: dependency analysis, call-graph-based analysis of library use augmented with hand-annotated library-specific sinks, and automated remediation. We illustrate the need for dynamic analysis in each of these tasks, using package manager integrations to identify dependencies and instrumentation to build dynamic call graphs. We also describe a novel means of composing the dynamic and static call graphs together with precomputed call chains, making the analysis modular in the library dependencies of the application. This significantly improves the ability to find vulnerable methods.

In future, we hope to focus on automated remediation, for example by performing transitive dependency upgrades, or optimizing upgrades by pruning redundant dependencies or reacting to dependency conflicts [12].

References

[1] Foo, Darius and Chua, Hendy and Yeo, Jason and Ang, Ming Yi and Sharma, Asankhaya, “Efficient static checking of library updates”, 2018.

[2] Foo, Darius and Ang, Ming Yi and Yeo, Jason and Sharma, Asankhaya, “SGL: A domain-specific language for large-scale analysis of open-source code”, 2018.

[3] Zhou, Yaqin and Sharma, Asankhaya, “Automated identification of security issues from commit messages and bug reports”, 2017.

[4] Dean, Jeffrey and Grove, David and Chambers, Craig, “Optimization of object-oriented programs using static class hierarchy analysis”, 1995.

[5] Sundarespan, Vijay and Hendren, Laurie and Razafimahefa, Chrislain and Vallée-Rai, Raja and Lam, Patrick and Gagnon, Etienne and Godin, Charles, “Practical virtual method call resolution for Java”, 2000.

[6] S. Raemaekers and A. van Deursen and J. Visser, “Semantic Versioning versus Breaking Changes: A Study of the Maven Repository”, 2014.

[7] Kula, Raula Gaikovina and German, Daniel M and Ouni, Ali and Ishio, Takashi and Inoue, Katsuro, “Do developers update their library dependencies?”.

[8] Mirhosseini, Samim and Parnin, Chris, “Can automated pull requests encourage software developers to upgrade out-of-date dependencies?”, 2017.

[9] Shahin, Mojtaba and Babar, Muhammad Ali and Zhu, Liming, “Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices”, 2017.

[10] Z. Xing and E. Stroulia, “API-Evolution Support with Diff-CatchUp”, 2007.

[11] Henkel, Johannes and Diwan, Amer, “CatchUp!: Capturing and Replaying Refactorings to Support API Evolution”, 2005.

[12] Wang, Ying and Wen, Ming and Liu, Zhenwei and Wu, Rongxin and Wang, Rui and Yang, Bo and Yu, Hai and Zhu, Zhiliang and Cheung, Shing-Chi, “Do the dependency conflicts in my project matter?”, 2018.

[13] Jiang, Lingxiao and Misherghi, Ghassan and Su, Zhendong and Glondu, Stephane, “Deckard: Scalable and accurate tree-based detection of code clones”, 2007.

[14] Sabetta, Antonino and Bezzi, Michele, “A Practical Approach to the Automatic Classification of Security-Relevant Commits”, 2018.

[15] Grove, David and Chambers, Craig, “A framework for call graph construction algorithms”, 2001.

[16] Rountev, Atanas and Kagan, Scott and Gibas, Michael, “Static and dynamic analysis of call chains in Java”, 2004.

[17] Clapp, Lazaro and Anand, Saswat and Aiken, Alex, “Modelgen: mining explicit information flow specifications from concrete executions”, 2015.

[18] Lauinger, Tobias and Chaabane, Abdelberi and Arshad, Sajjad and Robertson, William and Wilson, Christo and Kirda, Engin, “Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web”, 2018.

[19] Livshits, Benjamin and Sridharan, Manu and Smaragdakis, Yannis and Lhoták, Ondřej and Amaral, J Nelson and Chang, Bor-Yuh Evan and Guyer, Samuel Z and Khedker, Uday P and Møller, Anders and Vardoulakis, Dimitrios, “In defense of soundiness: a manifesto”, 2015.

[20] Smaragdakis, Yannis and Balatsouras, George and Kastrinis, George and Bravenboer, Martin, “More sound static handling of Java reflection”, 2015.

[21] Dufour, Bruno and Ryder, Barbara G and Sevitsky, Gary, “Blended analysis for performance understanding of framework-based applications”, 2007.

[22] Dufour, Bruno and Ryder, Barbara G and Sevitsky, Gary, “A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications”, 2008.

[23] “OWASP Top Ten Project”.

[24] Kula, Raula Gaikovina and Ouni, Ali and German, Daniel M and Inoue, Katsuro, “On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem”, 2017.

[25] Haefliger, Stefan and Von Krogh, Georg and Spaeth, Sebastian, “Code reuse in open source software”, 2008.

[26] Ponta, Serena Elisa and Plate, Henrik and Sabetta, Antonino, “Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software”, 2018.

[27] Millar, Stuart. “Vulnerability Detection in Open Source Software: The Cure and the Cause.” (2017).

[28] Lhotk, Ond. “Comparing call graphs.” 2007.

[29] Tip, Frank. “Infeasible paths in object-oriented programs.” (2015)

[30] Jalan, Rohit, and Arun Kejariwal. “Trin-trin: Whos calling? a pin-based dynamic call graph extraction framework.”

[31] Ali, Karim, and Ondej Lhotk. “Application-only call graph construction.” 2012.