Scalable Language Agnostic Taint Tracking Using Explicit Data Dependencies

Sedick David Baker Effendi
dbe@sun.co.za
Stellenbosch University
Stellenbosch, South Africa

Xavier Pinho
StackGen
San Ramon, California, United States

Andrei Michael Dreyer
Fabian Yamaguchi
Whirly Labs
Cape Town, South Africa

Abstract

Taint analysis using explicit whole-program data-dependence graphs is powerful for vulnerability discovery but faces two major challenges. First, accurately modeling taint propagation through calls to external library procedures requires extensive manual annotations, which becomes impractical for large ecosystems. Second, the sheer size of whole-program graph representations leads to serious scalability and performance issues, particularly when quick analysis is needed in continuous development pipelines.

This paper presents the design and implementation of a system for a language-agnostic data-dependence representation. The system accommodates missing annotations describing the behavior of library procedures by over-approximating data flows, allowing annotations to be added later without recalculation. We contribute this data-flow analysis system to the open-source code analysis platform Joern, making it available to the community.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Formal software verification.*

Keywords: static analysis, taint analysis, code property graph, data flow

1 Introduction

Continuous integration and deployment [6] are now standard in many organizations [9], but achieving similarly continuous vulnerability detection without slowing down the release process remains an ambitious goal. Vulnerability discovery techniques such as symbolic execution [e.g. 15, 19] or fuzz testing [e.g. 2, 16] fall short in this environment, as they assume a relatively static target. Expensive state exploration, however, stands in direct conflict with the need for quick feedback in modern pipelines.

Researchers have explored using graph databases to store and process whole-program representations of code [21]. In this context, explicit data-dependence representations have proven particularly useful in vulnerability discovery [14], as they can model a wide range of taint-style vulnerabilities, including command injections, file inclusion vulnerabilities, and cross-site scripting (XSS) vulnerabilities. These representations facilitate combining taint propagation information with syntactic and control-flow information to identify vulnerable code [26] and enable automated processing using graph-based machine learning algorithms [3]. However, their accuracy hinges on knowing the taint propagation semantics of all methods.

We present and implement a taint-tracking strategy based on a whole-program data-dependence representation that can be incrementally updated as knowledge about the semantics of external libraries becomes available, avoiding full recomputation when adding new annotations. We contribute our resulting work to an existing open-source code analysis platform, Joern [11], and make it available to the research community. We evaluate the efficiency of our analysis on Java, Python, and JavaScript programs.

2 Background

Consider a simple example of data flow that spans external calls to motivate our approach. Listing 1 defines two methods, `foo` and `bar`. The `foo` method obtains an object `u` from an external source (`Source.getValue`), creates a new object `v`, and calls `u.transform(v)` to produce `result`, which is then passed to `bar(result, v)`. The `bar` method calls an external method (`Sink.addValue`) on both its parameters. We want to know whether data from the source, i.e., the return of `Source.getValue`, can reach the first argument of `Sink.addValue`. This means checking if a call to `Source.getValue` *defines* a value w that is later *used* as w' in a call to `Sink.addValue`(w'). In other words, is w' obtained through a series of transformations from w?

Listing 1. Sample Java code with methods `foo` and `bar` that call external methods `getValue`, `isPrivileged`, `addValue`, and `transform`.

```
public class Example {
  public static void foo() {
    Obj u = Source.getValue();
    Obj v = new Obj();
    if (Config.isPrivileged()) {
      Obj result = u.transform(v);
      bar(result, v); // internal
    }
  }
  static void bar(Obj x, Obj y) {
    Sink.addValue(x); // sink
    Sink.addValue(y); // sink
  }
}
```

This perspective on code as operations on variables for which arguments are *used*, *defined*, or *used* and *defined* within a method can be expressed via a *data dependence graph*. Originally developed for program slicing [5], this graph contains edges from nodes describing operations that define a variable to those that use it without prior redefinitions.

When no information is available about external methods, an analyzer has two options: assume the calls have no effect or assume they taint everything. With the former approach, we obtain an under-approximated graph where no data dependencies are established between the external method calls. With the latter approach, we obtain an over-approximated graph with spurious data dependency paths that may not reflect actual taint propagation.

Compared to the under-approximated graph, the over-approximated graph offers the advantage that the possible data dependency between `result` at the call to `bar` and its occurrence at the call to `Sink.addValue` is indicated by a path in the data dependence graph. Similarly, the potential re-definition of `u` or `v` introduced by the call to `transform` is visible due to the inability to traverse from the node of `bar` to that of `Sink.addValue` without passing through that of `transform`. This is an example of a transitive data dependency, where a dependency from one variable to another is due to a chain of intermediate steps or functions [10].

To deal with these transitive dependencies, Horwitz et al. [10] proposes an elegant extension of intraprocedural data dependence graphs, which they refer to as *system dependence graphs*. In the system dependence graph, separate nodes for input and output arguments are introduced, and transitive dependencies are encoded via direct edges from input to output nodes. This compresses transitive dependencies onto a representation that only perceives local (non-transitive) dependencies.

Nonetheless, the idea of maintaining a representation of data dependencies that is independent of transitive data dependencies—and that therefore does not need to be recalculated as new information about external methods becomes available—forms the intellectual basis for our approach. This fits our scenario well, where the behavior of external methods may be characterized in greater detail by the user over time.

3 Design

The Joern [11] code analysis platform is extended with a data-flow engine. Joern's language frontends and standard stages generate a unified abstract syntax tree (AST), control flow graph (CFG), and control dependence graph (CDG), forming a near-complete *code property graph (CPG)* [26]. The data-flow engine provides the necessary primitives to construct the intermediate data dependence graph (DDG) for a full CPG. It includes a querying engine to determine flows on the fly for specified sources and sinks. This scheme is a *may* analysis that identifies flows from sources to sinks, considering user-provided semantics of external methods. The following sections describe the design and implementation in greater detail.

3.1 Data-Dependence Representation

The data-dependence representation is based on program dependence graphs (PDGs) [5] (see Figure 1). A problem

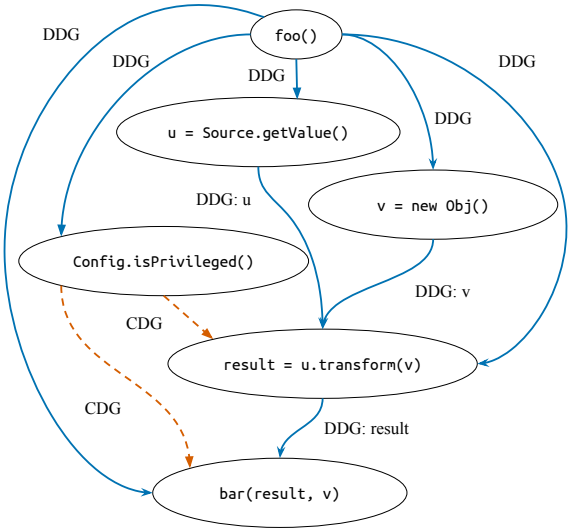

Figure 1. The program dependence graph of the code in Listing 1. Edges are labelled as belonging to either the control dependence graph (CDG) or the data dependence graph (DDG).

with this approach is that a method's data-dependence representation is only precise if the semantics of all its transitive callees are known [10]. As [10] furthermore shows, summary edges can encode these semantics and be calculated in polynomial time for callee methods available at analysis time. However, for external library methods (with code unavailable at compile time), the user must provide semantics, or they are assumed to be unknown. In this case, it is assumed that all input parameters may taint all output parameters to safely overapproximate the data flow. Palepu et al. [18] find success in dynamically generating program summaries for external library code as data and control dependencies between inputs and outputs of calls to external procedures. The authors acknowledge that these summaries can introduce unsoundness and imprecision; however, the performance gains may outweigh these costs. Toman and Grossman [24] explores how library code may bring along many transitive dependencies, and a resulting summary for a method may require referencing indirect flows to other functions. The related efforts in summarising external code suggest difficulties in how granular these could be in a language-agnostic

approach, and one must accept the inherent imprecision and unsoundness introduced by using such an approach.

To address these challenges, our data-flow engine maintains a stable data-dependence representation as users refine method semantics. It achieves this by treating all callees as external with unknown semantics, over-approximating data dependencies at each call site. Unlike the exploded supergraph in the classical IFDS framework [20], this approach does not rely on hard-coded semantics. While this introduces invalid paths, i.e. paths that are not valid in any runtime execution, they are discarded at query time. As a result, adding such summaries is not required to discover additional flows but helps eliminate false positives.

In Figure 1, we note that a call to `transform` is crucial in determining which paths from the `Source` to `Sink` classes are valid. As shown in Listing 2, we can describe the valid flows for `transform` to have several outcomes. To define a semantic, one must supply the method's full name, followed by a list of flows between arguments annotated by their positional index and/or argument name, for languages that support named arguments.

Certain positional indexes denote special cases. Such special cases are the return of a call, as index "`-1`", and the receiver as index "`0`", which denotes the object to which the method is bound. Any unspecified flows will be interpreted as *killed* or *sanitized*, i.e., no flow exists between the input and output node. Thus, we need to be explicit where flows are not killed, e.g., `0->0`. As this may become tedious for methods with many arguments, several special flow objects in the programmatic API provide shorthand ways to define common cases.

To explain how one can define these semantics, we detail the examples from Listing 2. The first parameter is sanitized (and thus omitted), while the receiver is not and propagates to the return value, defining flows `0->0` and `0->-1`. Next, we modify the semantics so that the receiver now taints the non-sanitized first parameter, resulting in flows `1->1` and `0->1`.

Listing 2. An example of user-supplied semantics for a call to `transform`.

```
/* E.g.1: Argument 1 is sanitized, receiver flow
 propagates to the return value */
"Obj.transform:Obj(Obj)" 0->0 0->-1

/* E.g.2: Receiver taints argument 1 */
"Obj.transform:Obj(Obj)" 0->0 1->1 0->1
```

Semantics can be written manually or programmatically. One can use heuristics, data-driven approaches, or the data-flow engine to programmatically generate and load new semantics on the fly until one needs to run a data-flow query.

3.2 Identifying data-flows

With the data-dependence representation in place, the next step is determining data flows based on user-provided queries. As is true for many other taint analysis systems [e.g., 1, 7, 22], our query consists of a set of sources and a set of sinks, and it is our goal to determine all source-sink pairs for which a flow from source to sink exists, along with a sample flow. However, as the data-dependence representation does not have hard-coded semantics, a query also includes a set of semantics for external library methods, as we allow the semantics of library methods to change.

Given such a query, the goal now is to calculate data flows in an algorithmically efficient manner that effectively uses multicore CPUs. To this end, an approach similar to Duesterwald et al. [4] is chosen. They answer queries incrementally, translating queries into tasks and deriving new tasks from the results of prior tasks at method boundaries. Using this approach, each task operates only within the boundaries of a method, such as `foo` or `bar` shown in Listing 1, and can be calculated independently and concurrently.

Taint analysis can be performed in forward and backwards modes: either traverses data-dependence edges from sources along the edge direction towards sinks or from sinks against the edge direction towards sources. In the following, only the taint analysis in the backwards direction is described, but the forward direction can be implemented analogously.

A (backwards) task is defined to be given by a start node, an already-known path from the start node to a sink node, the set of source nodes, and the set of semantics. Moreover, a positive integer that indicates analysis depth is stored, referring to the call-chain depth that the analysis explores.

To simplify notation, tasks are described only by pairing start nodes and paths from the start node to the sink; the result table and the analysis depth are assumed to be available. As the sources, sinks, and semantics remain constant throughout the processing of a query, it is assumed that they are available for reading via a globally shared object. With these simplifications in mind, for a given set of sinks $\mathcal{D}$, the initial set of tasks is given by $\{(d, [], 0) \mid d \in \mathcal{D}\}$, where $[]$ denotes an empty path, and 0 is the initial call depth.

These tasks are submitted to a work queue, with resulting paths pushed to the output queue. A result can either be *complete*, meaning it describes a flow from a source in $\mathcal{S}$ to a sink in $\mathcal{D}$, or it can be *partial*, meaning that it is a flow that may be part of a complete flow from a source to a sink. We fetch these results from the output queue, record complete results, and derive new tasks from each result. We note that tasks must also be created from complete results, as they may describe sub-flows of a larger complete flow. This procedure is carried out until all tasks have been evaluated and no more new tasks need to be submitted. At this point, all recorded results are returned.

A result is given by a path $p = ([(v_1, r_1), \ldots, (v_N, r_N)], k)$ where N is the path length, and for all i from 1 to N, v_i is a node, r_i is a Boolean, and k is the current call depth. For nodes that are arguments in method calls, the Boolean r_i indicates whether the associated method has been resolved in the process of generating the result (`true`) or whether resolving it has been deferred to a future task (`false`).

Translating results to new tasks. From a result p, new tasks are generated according to the following rules, shown by Algorithm 1. First, new tasks are only created if the new call depth is no larger than the maximum depth (line 3). If so, do not generate new tasks (line 4), resulting in partial tasks with no new dependent tasks. In this case, flows will be over-approximated for dependent callers of this result. This early termination is a form of widening to ensure termination, analogous to k-limiting [12]. Second, if the path begins with a parameter (line 8), we look up the set of corresponding arguments $\mathcal{A}$ (line 9) and generate the tasks $\{(a, p) \mid a \in \mathcal{A}\}$ (line 10). These corresponding arguments include positional or named arguments at call sites and the receiver of call sites referring to the parameter's method as a higher-order function. Finally, for the given path, all unresolved arguments are determined (line 11). For each unresolved argument, the tasks $\{(o, p) \mid o \in O\}$ (line 13) are generated from the set of associated formal output parameters O (line 12). If the argument is the actual return value of a call, the task (r, p) is also generated, where r denotes the corresponding formal return parameter. If the argument is a method reference, such as a closure, then the closure's return statement becomes a task (r_c, p), where r_c denotes the return statement of the closure.

Algorithm 1 Given a partial result p, generates new tasks from parameters and unresolved arguments using the call graph.

```
1:  procedure CreateTasksFromResult(p)
2:      (x, k) ← p        ▷ Extract the path x and call depth k
3:      if k + 1 >= k_max then
4:          return ∅
5:      end if
6:      (v, r) ← x[0]   ▷ Extract head node v and Boolean r
7:      T* ← ∅
8:      if IsParameter(v) then
9:          A ← GetArgsFromCallers(v)
10:         T* ← [(a, p, k + 1) for a ∈ A]
11:     else if IsArgument(v) and r is false then
12:         O ← GetUnresolvedOutArgsAndReturns(v)
13:         T* ← [(o, p, k + 1) for o ∈ O]
14:     end if
15:     return T*
16: end procedure
```

Solving tasks. Each task (s, p, k) is solved by a separate worker thread. Results are calculated by inspecting s alone and then determining results for all *valid parents*, that is, nodes with an outgoing data-dependence edge to s that is *valid* according to the semantics S.

The result for s is determined as follows. If the head of p is a source, the result is a complete path $(s, \text{false}) : p$, where ":" denotes an append operation. An additional partial path result is pushed if the source is a method parameter. This additional result allows Algorithm 1 to create a new task from this result and possibly find additional sources later. If the head of p is not from the source set but a method parameter, then $(s, \text{false}) : p$ is returned as the path for a partial result.

To determine edge validity, the edges from actual returns of method calls are discarded if the semantic value states that the call does not define the return argument. If s is not an argument, we return the remaining list of parents. If s is an argument, incoming edges from parent nodes that are not arguments are valid. In these cases, return a partial result and mark the result as unresolved. These cases either reflect an incomplete call graph or that the task depends on a partial task that was discarded for exceeding the maximum call depth. In either case, the outcome will be that the result is over-approximated, i.e., it is assumed that all of its arguments are both used and defined by a call to the method.

Validity of parents based on semantics. A parent node s_0 is connected directly to s via an outgoing data-dependence edge, but not all edges are valid according to the semantics. For parent nodes that are arguments, if s and s_0 are arguments of the same call site and the parent node is used while s is defined according to the semantics, the edge is valid. The edge is also valid if s and s_0 are arguments of different call sites, but s is used according to the semantics; otherwise, the edge is invalid. The data flows are over-approximated for methods without defined semantics.

Computing results for valid parents. In the following, we refer to Algorithm 2. For each valid parent (lines 3–4), whether a result exists in the table is determined (line 5), and if so, it is used to compute the result by determining the sub-path from the current parent node to the sink and appending p, followed by the parent s_0. Otherwise, the result is computed recursively; that is, we compute the results for $s_0 : p$. Upon collecting results for all parents and the head node, deduplicate and return (line 13).

Finally, the union of the results for p and its valid parents is returned. This result is stored in the result table as a cache.

4 Limitations

Operators such as assignments, arithmetic and field accesses are modeled as ordinary call nodes with a default set of semantics. Consequently, aliasing and the heap of data structures are not tracked. In the case of aliases, assignments will

Algorithm 2 Given a task (s, p, k), determine valid results for parents of s using the semantics and data-dependence representation.

```
1:  procedure ComputeResultsForParents(s, p, k)
2:      R* ← ∅
3:      for all s_0 in Out(s) do                ▷ Traverse DDG edges
4:          if IsValidEdge(s, s_0) then
5:              if s_0 ∈ R* then                ▷ Prepend known path
6:                  R* ← R* ∪ (R*[s_0] : p, k)
7:              else
8:                  r_0 ← IsOutputArg(s_0)
9:                  R* ← R* ∪ ((s_0, r_0) : p, k)
10:             end if
11:         end if
12:     end for
13:     return Deduplicate(R*)
14: end procedure
```

propagate flow, but only via weak updates. For data structures that use index accesses for arbitrary keys such as index values or keys in maps, the data-flow engine tracks these as "containers": if an internal member is tainted, then by the semantic definition, the container is tainted.

This leaves future work to make this analysis alias and object-sensitive. However, this imprecision may be attributed to the analysis's low overhead.

5 Evaluation

This evaluation aims to answer the following research questions: (**RQ1**) Is the system able to detect taint-style vulnerabilities effectively for multiple programming languages, and (**RQ2**) without analyzing library code? Finally, an essential property for data-flow analysis in the context of modern programs is (**RQ3**), i.e., how scalable is our analysis?

5.1 Method

We compare the precision of the data flow engine of Section 3 against two multi-language taint analyzers, Semgrep [22] and CodeQL [7]. The primary considerations for related work are that they are widely adopted, support multi-language taint analysis, and allow partial program analysis. Each tool is run on the same Java, JavaScript, and Python benchmarks, where partial and whole program analysis techniques are compared. We measure precision and recall using the F1-score and Youden's J index [27] (rewards higher specificity). We also recorded each tool's analysis runtime and memory usage to assess scalability. All experiments[1] were performed on a platform with a 6-core x86 CPU (3.4 GHz), 32 GB memory, and running Java 21.0.2.

[1]Reproducible at https://github.com/joernio/joern-benchmarks.

5.2 Dataset

Choosing a *well-suited* dataset for taint analysis is non-trivial, where we define well-suited as publicly available and providing a sink, source, and the outcome for any given test. Securibench Micro [17] meets these criteria for Java.

While Guarnieri et al. [8] mentions developing a JavaScript benchmark akin to Securibench Micro, the associated link is dead when writing. To this end, and as a contribution, we develop *securibench-micro.js*[2] as a JavaScript equivalent to Securibench Micro. For Python, such a benchmark is not readily available; however, an incomplete synthetic benchmark similar in spirit to Securibench Micro exists. As another contribution, this benchmark is completed and dubbed "Thorat" [23][3] after its original author.

When measuring scalability, however, none of the programs in the datasets above compares in magnitude to an industry-sized program. To address this shortcoming, we use Defects4j [13] and include a Python-inspired variant, namely BugsInPy [25]. While not intended for measuring taint analysis, they include real-world programs that test the scalability of a static analysis tool. The latest versions of each program of these datasets are obtained at the time of writing.

5.3 Determining a Suitable Analysis Depth

To justify a suitable value for k to be used by the Joern-based data-flow analysis, one must explore how different values for k affect the results. The results' figures for finding an appropriate value for k have been omitted here for brevity but can be found under Section A.

When measuring against the taint analysis benchmarks, each experiment runs for 10 iterations with user-defined semantics enabled. A significant variation in the J index and F score appears between $k \in [0, 3]$, followed by a slight increase in precision when $k = 8$ in Securibench Micro and securibench-micro.js. When observing the taint analysis wall-clock times for Defects4j and BugsInPy, for a subset of programs, the beginning of exponential complexity for runtime is observed from $k = 6$. Thus, to strike a balance between precision and recall while remaining practical, we determine that $k = 5$ is a safe value for k.

5.4 Taint Analysis

This section outlines Joern's performance with the presented data-flow engine, using a max call depth $k = 5$, for the three benchmarks against Semgrep and CodeQL.

Table 1 presents the results of partial taint analysis for each evaluated tool. Joern is assessed in two configurations: without user-defined semantics (Joern) and with manually specified semantics (Joern_{SEM}). Both configurations include operator semantics, but the latter incorporates additional,

[2]Available at https://github.com/DavidBakerEffendi/securibench-micro.js.
[3]Fork available at https://github.com/DavidBakerEffendi/thorat.

Table 1. Benchmark results on the partial program static taint analysis for Joern, Joern$_{SEM}$, Semgrep, and CodeQL.

Benchmark	Tool	TP	TN	FP	FN	J Index	F1 Score	Runtime (s)	Memory (GB)
Securibench Micro	Joern	119	17	36	17	0.196	0.818	**1.48±0.54**	0.33 ± 0.03
	Joern$_{SEM}$	118	36	17	18	**0.547**	**0.871**	1.74 ± 0.59	0.29 ± 0.02
	Semgrep	100	39	14	36	0.471	0.800	16.73 ± 0.57	**0.14±0.01**
	CodeQL	93	37	16	43	0.382	0.759	79.57 ± 1.05	1.26 ± 0.05
Thorat	Joern	29	22	12	11	0.372	**0.716**	0.91 ± 0.34	0.26 ± 0.02
	Joern$_{SEM}$	29	22	12	11	0.372	**0.716**	**0.77±0.25**	0.25 ± 0.02
	Semgrep	15	24	10	25	0.081	0.462	15.01 ± 0.41	**0.14±0.01**
	CodeQL	23	29	5	17	**0.423**	0.676	44.76 ± 0.69	0.97 ± 0.04
securibench-micro.js	Joern	93	18	26	24	0.204	0.788	6.94 ± 0.49	0.29 ± 0.02
	Joern$_{SEM}$	93	19	25	24	0.227	**0.791**	**6.88±0.47**	0.28 ± 0.02
	Semgrep	5	43	1	112	0.020	0.081	14.64 ± 0.57	**0.14±0.01**
	CodeQL	85	31	13	32	**0.431**	**0.791**	93.99 ± 1.98	1.31 ± 0.05

manually curated semantics for external procedure calls, thereby mitigating unnecessary false positives. Whole program analysis is also considered, where the datasets used by Table 1 are appended with their external dependencies, including transitive ones. The results of these experiments are omitted here but can be found under Section A.

5.5 Discussion

Semantic annotations reduce false positives in Securibench Micro, none in Thorat, and one in securibench-micro.js. This is likely due to Java being a more verbose language than Python and JavaScript, leading to the overtainting of more data flows if calls are left unconstrained. Similarly, operations on data structures in JavaScript and Python often use syntactic sugar present as operators, such as index or property accesses.

CodeQL is a reliable choice for analyzing dynamic languages when considering precision, and Semgrep generally falls short. From the evaluation, the Joern-based data-flow analysis is the most performant taint analysis in this evaluation. This analysis can identify the most vulnerabilities; however, this comes at the cost of additional false positives. While the user-defined semantics have been shown to reduce false positives without needing to rerun the analysis, the lack of precision for dynamic languages leaves room for future work.

For whole-program analysis, the Joern results reported fewer false negatives and, in some cases, fewer true positives. The cost of the whole analysis scaled the worst compared to the other candidates. However, it still ended up being the fastest tool in most cases. Compared to the partial-program analysis, a significant cost is incurred for a small precision gain, thus suggesting that the benefits of partial-program analysis outweigh the imprecision.

When comparing the results along the memory column, beyond the discrepancy of Joern performing worse in Java, the Joern-based approaches have a memory footprint far closer to that of Semgrep while being closer to CodeQL in precision. However, this figure suggests that CodeQL may scale better than Joern regarding memory consumption for sufficiently large programs.

Joern's precision for partial program analysis is serviceable and falls somewhere between Semgrep and CodeQL. The individual success of these tools supports the applicability of our work in real-world applications. The results, interpreted through the constraints of **RQ1** and **RQ2**, indicate that our tool effectively and efficiently performs partial-program static taint analysis across multiple programming languages. The results in all categories suggest that the answer to **RQ3** is that Joern is scalable enough for partial-program analysis of modern programs.

6 Conclusion

In response to the growing demand for performant vulnerability discovery in large systems, this paper presented a system capable of language-agnostic static taint analysis without direct access to external dependencies. By using simple annotations to model these dependencies, this system can answer taint analysis queries written with a high-level query language without having to re-analyze the dependencies.

References

[1] Eric Bodden. 2012. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the International Workshop on State of the Art in Java Program analysis*. Association for Computing Machinery, New York, NY, USA, 3–8.

[2] E. Bounimova, P. Godefroid, and D. Molnar. 2013. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proc. of the International Conference on Software Engineering (ICSE)*. IEEE Press, San Francisco, CA, USA, 122–131.

[3] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there

yet. *IEEE Transactions on Software Engineering* 48, 09 (2021), 3280–3296.

[4] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1997. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 6 (1997), 992–1030.

[5] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[6] Martin Fowler, Jim Highsmith, et al. 2001. The agile manifesto. *Software development* 9, 8 (2001), 28–35.

[7] GitHub, Inc. 2024. CodeQL (Version 2.19.2). https://codeql.github.com. Retrieved June 2024.

[8] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada). Association for Computing Machinery, New York, NY, USA, 177–187.

[9] Bill Holz and Mike West. 2019. Results Summary: Agile in the Enterprise (Updated). https://circle.gartner.com/Portals/.../Summary%20(updated).pdf. Retrieved July 2021.

[10] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 1 (1990), 26–60.

[11] Joern Community. 2024. Joern (Version 4.0.119). https://github.com/joernio/joern. Retrieved October 2024.

[12] Neil D Jones and Steven S Muchnick. 1979. Flow analysis and optimization of LISP-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. Association for Computing Machinery, San Antonio, Texas, 244–256.

[13] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. Association for Computing Machinery, New York, NY, USA, 437–440.

[14] Soheil Khodayari and Giancarlo Pellegrino. 2021. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In *Proc. of USENIX Security Symposium*. USENIX Association, Vancouver, B.C., 2525–2542.

[15] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[16] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 562–566.

[17] Benjamin Livshits. 2006. Securibench Micro. https://github.com/too4words/securibench-micro. Retrieved May 2024.

[18] Vijay Krishna Palepu, Guoqing Xu, and James A Jones. 2017. Dynamic dependence summaries. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 4 (2017), 1–41.

[19] Corina S Păsăreanu and Willem Visser. 2009. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer* 11, 4 (2009), 339–353.

[20] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of the Symposium on Principles of programming languages (POPL)*. Association for Computing Machinery, New York, NY, USA, 49–61.

[21] Oscar Rodriguez-Prieto, Alan Mycroft, and Francisco Ortin. 2020. An efficient and scalable platform for java source code analysis using overlaid graph representations. *IEEE Access* 8 (2020), 72239–72260.

[22] Semgrep, Inc. 2024. Semgrep (Version 1.95.0). https://semgrep.dev. Retrieved May 2024.

[23] Rajiv Thorat. 2022. Benchmark For Taint Analysis Tools Python. https://github.com/rajiv-thorat/benchmark-for-taint-analysis-tools-for-python. Retrieved May 2024.

[24] John Toman and Dan Grossman. 2017. Taming the static analysis beast. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:14.

[25] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. Association for Computing Machinery, New York, NY, USA, 1556–1560.

[26] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proc. of IEEE Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, USA, 590–604.

[27] William J Youden. 1950. Index for rating diagnostic tests. *Cancer* 3, 1 (1950), 32–35.

A Supplementary Evaluation Figures

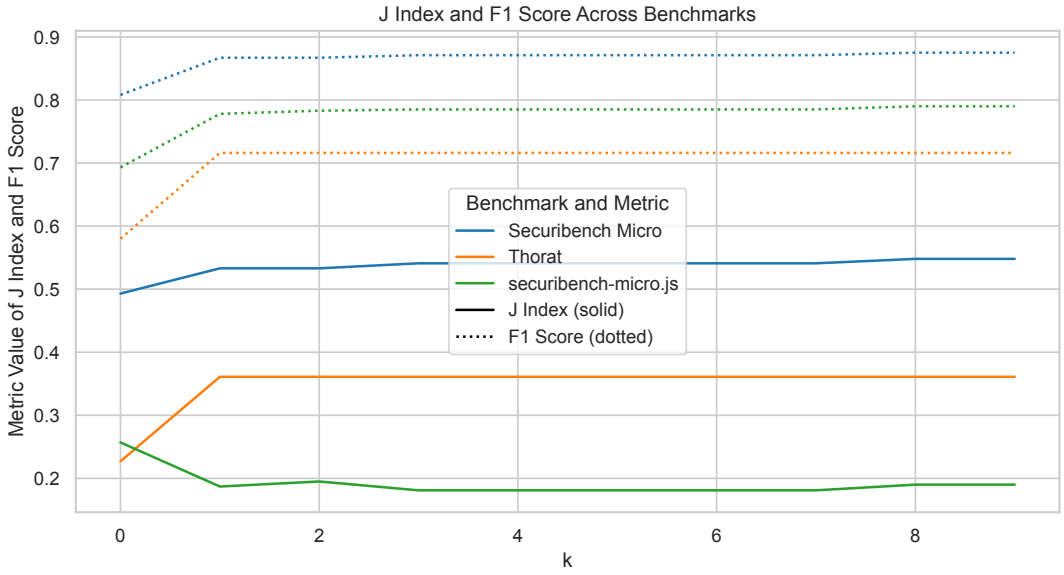

Figure 2. The results when exploring for an appropriate value for k using each taint analysis benchmark.

Table 2. Benchmark results on the whole program static taint analysis for Joern, Semgrep, and CodeQL. As Semgrep does not parse bytecode, the Java results are omitted.

Benchmark	Tool	TP	TN	FP	FN	J Index	F1 Score	Runtime (s)	Memory (GB)
Securibench Micro	Joern	113	37	16	23	**0.529**	**0.853**	**103.27±5.50**	1.91 ± 0.03
	Semgrep	-	-	-	-	-	-	-	-
	CodeQL	93	37	16	43	0.382	0.759	109.73±2.20	**1.53±0.03**
Thorat	Joern	27	23	11	13	0.351	**0.692**	**22.01±8.40**	0.90 ± 0.03
	Semgrep	15	24	10	25	0.081	0.462	27.09 ± 0.52	**0.16±0.01**
	CodeQL	23	29	5	17	**0.428**	0.676	65.96 ± 0.15	1.35 ± 0.05
securibench-micro.js	Joern	92	38	6	25	**0.650**	**0.856**	58.90 ± 3.31	1.19 ± 0.03
	Semgrep	5	43	1	112	0.020	0.081	**34.12±0.61**	**0.17±0.01**
	CodeQL	85	31	13	32	0.431	0.791	134.47 ± 1.54	1.83 ± 0.02

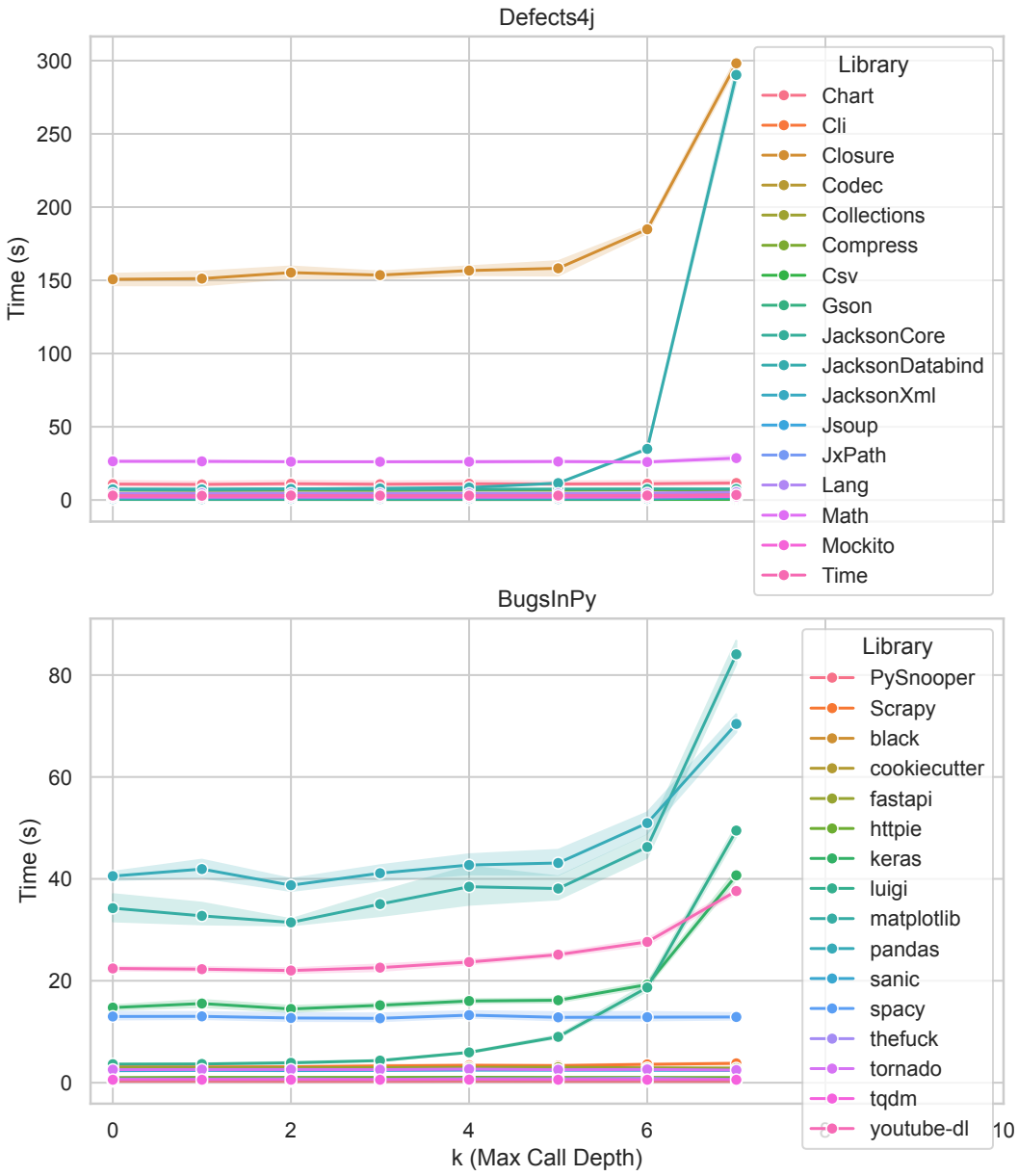

Figure 3. The performance of creating a code property graph and performing taint analysis on the programs of Defects4j and BugsInPy for varying values of $k \in [0, 7]$.