Compositional Taint Analysis for Enforcing Security Policies at Scale

Subarno Banerjee
Amazon Web Services

Siwei Cui
Texas A&M University

Michael Emmi
Amazon Web Services

Antonio Filieri
Amazon Web Services

Liana Hadarean
Amazon Web Services

Peixuan Li
Amazon Web Services

Linghui Luo
Amazon Web Services

Goran Piskachev
Amazon Web Services

Nicolás Rosner
Amazon Web Services

Aritra Sengupta
Amazon Web Services

Omer Tripp
Amazon Web Services

Jingbo Wang
University of Southern California

ABSTRACT

Automated static dataflow analysis is an effective technique for detecting security critical issues like sensitive data leak, and vulnerability to injection attacks. Ensuring high precision and recall requires an analysis that is context, field and object sensitive. However, it is challenging to attain high precision and recall and scale to large industrial code bases. Compositional style analyses in which individual software components are analyzed separately, independent from their usage contexts, compute reusable summaries of components. This is an essential feature when deploying such analyses in CI/CD at code-review time or when scanning deployed container images. In both these settings the majority of software components stay the same between subsequent scans. However, it is not obvious how to extend such analyses to check the kind of contextual taint specifications that arise in practice, while maintaining compositionality.

In this work we present *contextual dataflow modeling*, an extension to the compositional analysis to check complex taint specifications and significantly increasing recall and precision. Furthermore, we show how such high-fidelity analysis can scale in production using three key optimizations: (i) discarding intermediate results for previously-analyzed components, an optimization exploiting the compositional nature of our analysis; (ii) a scope-reduction analysis to reduce the scope of the taint analysis w.r.t. the taint specifications being checked, and (iii) caching of analysis models. We show a 9.85% reduction in false positive rate on a comprehensive test suite comprising the OWASP open-source benchmarks as well as internal real-world code samples. We measure the performance and scalability impact of each individual optimization using open source JVM packages from the Maven central repository and internal AWS service codebases. This combination of high precision, recall, performance, and scalability has allowed us to enforce security policies at scale both internally within Amazon as well as for external customers through integrations into multiple external AWS cloud services.

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Theory of computation → Program analysis**; • **Computer systems organization → Cloud computing**.

Keywords

software security, taint analysis, static analysis in the cloud

1 INTRODUCTION

Enterprises enforce a wide range of security policies on software applications to detect potential vulnerabilities [8, 20, 22], data leaks [13, 21] , information flow policy breaches [10], etc. A common route to enforcing these policies is statically analyzing code and configurations and issuing warnings to the user either early on in software lifecycle, e.g., during code reviews [2, 6, 7], or analyzing deployed artifacts like containerized applications [3, 9] and issue high-severity warnings [18, 19]. Irrespective of the stage at which static analysis tools are deployed, it is essential that these tools have a low false positive rate to minimize the effort and time required to investigate these warnings, and a low false negative rate in order to ensure high coverage w.r.t. the properties being checked. Typically, achieving these goals is at odds with scaling to analyzing millions of lines of code in industry-scale applications [31].

Tracking dataflow from *sources* to *sinks* can detect a large class of security vulnerabilities i.e., it can report dataflow from APIs where user-controlled or "tainted" data enters the application to where the data reaches security-sensitive endpoints. A vast amount of research exists in scaling static *taint analysis* such as demand-driven approaches that follow a "start anywhere" in code [47, 49], modular bottom-up analysis [35], and bi-abduction based analysis [30]. In this paper, we describe CompTaint, a compositional taint analysis for Java code that is deployed internally within AWS and externally as part of two cloud-based services–Amazon CodeGuru Reviewer [2] and Amazon Inspector [18].

Design Choice CompTaint implements a field, object and context sensitive compositional heap analysis following the approach in [35, 41]. We extend this heap analysis to a compositional *taint* analysis using the approach in [36]. We made the design decision to focus on a compositional analysis because it unblocks optimizations that are key to our two main use cases: code-review integration on internal code bases and in CodeGuru Reviewer [2], and container scanning as part of Amazon Inspector [18]. A compositional analysis typically involves computing a generalizable summary of a program component that can be applied to multiple contexts, e.g., computing analysis summary of a method that can be applied to different calling contexts to obtain the *context-sensitive* analysis states at call sites. This is important when deploying an analysis that posts recommendations at code review time as most of the code stays the same between commits. Reusing the analysis results from a previous scan for unchanged components ensures a fast turnaround. Furthermore, code artifacts deployed in containers often consist of many open source libraries that do not change between deployments. Precomputing analysis results for such libraries greatly reduces analysis time.

The benefits of such a modular analysis (avoiding repeated reanalysis of components, e.g., per usage context, and analyzing independent components in parallel) are well-established and have been discussed by previous work [40, 41]. In this paper, we present three orthogonal performance optimizations that were key to deploying this analysis in production including discarding the intermediate state, an optimization intrinsic to the compositional nature of the analysis. Furthermore, we present an extension to the taint analysis in order to verify complex taint specifications that arise in practice, while maintaining the compositional nature of the analysis resulting in a significant improvement in precision.

Soundness and Precision A lot of work in the literature explores the impact of memory abstractions [37] and design choices around context, flow, and field sensitivities [46, 48, 49] on the precision of static analyses. While precisely tracking dataflow from sources to sinks is indeed important to maintaining a low false positive rate, an equally important aspect that has received significantly less attention is the problem of precisely identifying sources and sinks in source code. With the notable exception of CodeQL [6], many taint specifications in the literature simply list a set of APIs of interest [17, 43] that are marked as sinks or sources. However, in practice the specific behavior of these APIs that determines whether they are sinks, sources or sanitizers depends on the context in which they are called. For example, the Java `Cipher` class will either perform encryption (behave as sanitizer) or decryption (behave as source) depending on how it was initialized.

We found that in addition to precisely tracking dataflow, the precision of the analysis significantly depends on the accuracy of identifying program locations matching such contextual taint specifications. The context could be constant values passed to certain APIs as in the `Cipher` example, sequences of API calls to define sources or sinks, and such. To address this concern while maintaining compositionality, we developed a novel *speculative context resolution* technique integrated into the compositional taint analysis. This technique resulted in a reduction in the false positive rate of CompTaint by 9% on average on a large corpus of real-world examples as motivated in § 2.1.

Steps before Production To ascertain that CompTaint is *production-ready*, we evaluated it on the OWASP$^+$ benchmark [14] with ground-truth, conducted shadow reviews on datasets without ground-truth and iterated on adding features in the analysis to address recall and precision. Once the analysis achieved best-in-class OWASP score among competing tools and a stipulated high acceptance rate in its internal deployment ($<$ 20% false positives for all its information flow rules), we focused on scaling the analysis to larger analysis targets followed by large number of analysis targets. A target is any analyzable artifact. For example, a target could be a JAR file from build artifacts of a code repository or even a collection of JAR files including the runtime dependency closure of a set of code repositories. Before deploying in production, we evaluated the analysis on datasets representative of two deployment scenarios: (a) a small number of code artifacts as target. This represents the deployment in CI/CD on code reviews alongside other cloud-based SAST tools that runs in AWS [1, 34, 42]. (b) large dependency closures of a code artifact, typically containing hundreds of code artifacts, representing analysis of containerized applications running in the cloud [18, 19]. Out of the box, the analysis did not scale to the single-target deployment scenario above.

Contributions In this paper, we first describe CompTaint's compositional taint analysis emphasizing key features that allowed it to meet the recall and precision bar inside AWS. Specifically:

- We developed an encoding on top of our abstraction of the heap to perform a **compositional taint analysis**, including a novel **speculative context resolution** technique to identify contexts around sources, sinks, and sanitizers, which significantly increased the precision of our analysis while retaining its compositional formulation.

Next, we describe a set of optimizations that made it possible to scale CompTaint to large industry-scale applications in its production deployment:

- **Discarding intermediate analysis state**: we leverage compositionality of our analysis to discard a large fraction of the abstract state for analysis components that are already summarized. We measure its effect and show how it favorably impacts CompTaint.
- **Analysis scope reduction**: we design a light-weight *scope-reduction* analysis that prunes entry-points into the program that if analyzed could not produce a security vulnerability given a set of input taint specifications. This optimization elides the analysis of a sizeable amount of code significantly reducing analysis complexity without compromising soundness.
- **Caching invocation models**: we implement caching of applicable taint specifications matching invocation sites of taint relevant APIs. We show that this substantially reduce the time for the scope-reduction analysis and the taint analysis.

Evaluation In this paper, we evaluate CompTaint on 20 artifacts from Maven Central [12] and code artifacts from 4 external AWS services. In order to present results comparable to CompTaint's deployment in Amazon Inspector where it scans large containerized applications, we create analysis targets by generating code artifacts of the dependency closures of 500 Maven Central repositories and use a sampling methodology to select the closures. Likewise, to measure CompTaint's performance on scans of industry-scale cloud services, we analyze the dependency closures of AWS services starting from a few known root repositories. We measure the effect of each optimization on the above datasets and describe how these techniques underpinning our analysis turned out to be critical in production. In order to evaluate the efficacy of *speculative context resolution* we evaluate CompTaint on a dataset of injection vulnerabilities. Further, to establish that the baseline analysis with context resolution, before the performance optimizations, has state-of-the-art recall and precision, we evaluate CompTaint on the dataset that includes OWASP [14], an industry standard for benchmarking security properties, among other real-world code examples.

```
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    log.info(cipher.doFinal(secretText));
```

Code 1: Cipher sanitizer for sensitive data leak.

Deployment CompTaint is deployed internally at Amazon integrated with the code-review system. CompTaint runs an ensemble of checks, and automatically posts recommendations on code reviews based on its findings. Developers have the option of marking recommendations as useful or not useful. Based on this developer feedback, CompTaint has an average acceptance rate of > 80% [1]. CompTaint is also deployed externally as part of an AWS service called Amazon Inspector [4, 18] and Amazon CodeGuru Reviewer [2]. CompTaint powers Amazon Inspector to execute high-fidelity scans of containerized AWS Lambda [5, 18] functions.

2 MOTIVATION

In this section, we present several motivating examples to illustrate the kind of complex contextual taint specifications that arise in practice. Additionally, we motivate the need for additional performance optimizations by showing empirical results on running the baseline analysis on benchmarks from Maven Central [12] and code artifacts from four external AWS services using the methodology described in § 6.

Traditionally, taint tracking tools [17, 43] specify sources, sinks and sanitizers at the API level by matching against a given method signature. However, in practice whether a given API acts as a source, sink or sanitizer often depends on the context. Consider Code 1: whether the `Cipher.doFinal` method performs encryption and acts as a sanitizer for sensitive data, depends on whether the `Cipher` class was initialized with the `Cipher.ENCRYPT_MODE` option. It is not safe to assume that any call of `Cipher.doFinal` performs encryption. As another example, consider checking whether Code 2 is vulnerable to cross-site scripting: we want to ensure that attacker controlled data does not reach the `HttpServletResponse.getWriter().write` method. Note that the `HttpServletResponse.getWriter()` method returns a `PrintWriter`. Simply matching on the `PrintWriter.write` method signature results in many spurious findings.

Finally, Code 3 shows a more complex example of object deserialization using an `XStream` [23] instance typically used to serialize and deserialize objects in XML and JSON formats. The simplicity of usage of `XStream` comes with the cost of exploitability. It has been exploited by researchers and adversaries to inflict remote command execution and denial-of-service attacks [24]. In Code 3, a new `URL` is created from untrusted external input on line 6 and `InputStream` created from the `URL` is later deserialized using `XStream` on line 7. The `XStream` library now provides methods to allow list *trusted* types using `allowTypes`. Line 14 shows this potential mitigation to the vulnerability by calling `safeConfigure` before `readUrl` on line 3. Observe that tainted data–variable `str`–still flows into the sink on line 11. The *context* that the analysis must capture is associated with variable `xs` and not the tainted variable `str`, and the sink `fromXML()` is neutralized due to *safe* state of the `xs` object. This example also illustrates that precisely tracking the context requires an inter-procedural analysis.

```
void process(HttpServletRequest req,
             HttpServletResponse rsp) {
    String param = req.getParameter("val");
    rsp.getWriter().write(param); // XSS sink
    PrintWriter writer = new PrintWriter();
    writer.write(param);          // not a sink
}
```

Code 2: Cross-site-scripting (XSS) sink.

```
1  void parse(String url) {
2      XStream xs = new XStream();
3      readUrl(url, xs);
4  }
5  void readUrl(String url, XStream xs) {
6      InputStream in = new URL(url).openStream();
7      Snapshot obj = (Snapshot) deserialize(xs, in);
8  }
9  Object deserialize(XStream xs, InputStream in) {
10     byte[] str = in.readAllBytes();
11     return xs.fromXML(str);
12 }
13 void safeConfigure(XStream xs) {
14     xs.allowTypes(new Class[] {Snapshot.class,
        Envelope.class})
15 }
```

Code 3: XML external entity sink.

2.1 Contextual Taint Specifications

Not taking the context into account, and matching the taint specification at the API level results in a precision loss of 9.85% on average and as much as 50% on certain vulnerability categories (§ 6.1: Table 1 shows detailed results).

Compositionality in Contextual Data-flow In order to accurately identify the context around matching taint specifications on program values that are not tainted but relevant to context, the analysis could use other sub-analyses to identify the sources, sinks, and sanitizers precisely, either apriori or synchronously with the main analysis. These sub-analyses could use light-weight, local analyses to identify these contexts imprecisely, use demand-driven heavy-weight inter-procedural precise analyses such as constant propagation, type-state, and complex-value flow analysis. We overview the compositional taint analysis that powers CompTaint in § 3. Our design resolves these contexts in the same pass integrated with the compositional taint analysis. First, in the use cases we encountered the context spanned across large depths of inter-procedural data-flow obviating local lightweight analysis. Second on-demand analysis does not scale beyond a bounded depth of inter-procedural flow in practice due to an exponential number of recursive queries [34], and it's challenging to reuse partial analysis results due to new contexts that renders the partial summaries invalid [28]. Third, we wanted to keep the compositional formulation such that the analysis remains compatible with our goals of leveraging reusable summaries of already analyzed program components. Fourth, as § 3 will detail, the computation of the abstraction of the heap computed for every *strongly connected component* (SCC) in the program is expensive and is prohibitive to be repeated for flow of different types through the heap such as constants, API calls for context, as well as other metadata for the taint analysis ruling out the possibility of using multiple sequential analyses.

2.2 Practical Challenges: Speed and Scale

Before we describe the techniques in § 3 underpinning the high precision in contextual data-flows (results in § 6.1), we note that this

[1] We cannot disclose exact details around numbers of recommendations and their validity for AWS internal subjects, so we present conservative numbers here.

appealing result initially came with practical setbacks. We performed offline experimentation on initial versions of our evaluation datasets (see § 6.2.1) before deploying CompTaint in production. We ran the baseline analysis without the performance optimizations in § 4 on the dependency closures of repositories from Maven Central [12] and dependency closures of target repositories of 4 AWS services. We observed that 19.5% out of 82 Maven dependency closures timed out with a time limit of 1 hr; the logs showed on average 57.5% of the whole program was not analyzed on the Maven closures computed based on number of SCCs processed. This raises the following question: *Is there a path to efficiency without sacrificing accuracy?* To answer the question we started investigating on three main areas: (a) Given a set of taint specifications S, does the analysis need to analyze all components in the global callgraph G built from all the target code artifacts to retain soundness and precision? (b) Are there performance bottlenecks in the analysis? (c) Are there any symptoms of memory bottlenecks, and if yes, can we address the issues leveraging the compositional analysis design?

3 COMPOSITIONAL ANALYSIS

In this section we give a high level overview of CompTaint 's compositional analysis algorithm. To handle heap aliasing compositionally, we use the approach described in [41] to compute context-independent summaries that are agnostic to the input heap. To achieve compositional taint tracking, we extend the compositional heap summaries of [41], to taint summaries by taking the approach presented in [36]: heap effects in summaries are extended with taint effects (§ 3.2). Taint effects capture how the tainted specification applies to code e.g. whether a heap location contains tainted data coming from a source, or whether it flows into a sink.

At a high level, CompTaint considers each method in the program as a component, i.e., the unit of composition. For each method, CompTaint computes its *effects* using the effects computed for the methods it calls. § 3.1 describes the definition of a component in presence of recursion. Effects, which we describe in § 3.2, capture dataflow relevant behavior, including heap accesses, and taint sources and sinks, among other analysis state. CompTaint computes methods' effects in dependency order, i.e., callees before callers. The dependency order is determined from the *call graph*, which we describe in § 3.1. CompTaint computes the effects of each method by iterating over the effects of its statements. Since the call graph may be cyclic, and individual methods can contain loops, CompTaint computes the limits of these iteration sequences to ensure methods' effects capture all possible behaviors. We guarantee termination by ensuring these limits have fixed points by applying abstractions to approximate effects [32].

3.1 Component Dependency Order

To determine the dependency order between program components, CompTaint first computes a whole-program call graph. Technically, the call graph provides a mapping from program statements that might invoke some method, i.e., *call sites*, to methods that are potentially invoked, i.e., *call targets*. We obtain a dependency graph among methods by identifying call sites with their enclosing methods. However, this dependency graph may be cyclic, due to either recursion or call-graph imprecision. To obtain the desired dependency order among components, we compute the strongly connected components (SCCs) of the method dependency graph. CompTaint then considers each SCC as one single component, and computes components' effects in SCC dependency order.

To achieve an adequate balance of precision and scalability, CompTaint computes call graphs via the variable type analysis (VTA) algorithm [50] implemented by SPARK [39]. This algorithm utilizes an inexpensive yet whole-program context-, flow- and object-insensitive "pointer analysis" using a data structure called the *type-propagation graph* or *pointer assignment graph (PAG)*. Graph nodes represent program variables, and edges represent assignments. Program types are seeded to their corresponding graph nodes at allocation sites, and propagated across graph edges to the nodes corresponding to call-site receivers. We obtain the resulting call graph by collecting methods' implementations for the types propagated to each call site as potential targets. We achieve this in linear time by computing and propagating types over the strongly connected components of the PAG.

3.2 Compositional Effects

CompTaint computes effects capturing the behaviors relevant to dataflow analysis. These effects include whether a given program value originated from a dataflow source, reached a dataflow sink, or was processed by a dataflow sanitizer. Since these effects are semantic properties relative to the policy being enforced, their specifications are provided as input rather than hard-coded into the analysis. CompTaint consumes such specifications as models that apply to program statements. For example, models can specify that sink effects are applied to the input arguments of SLF4J logging API calls, or that a sanitizer effect is applied to the return value of an application-specific sanitizer method.

Because CompTaint analyzes each component in isolation, we must capture these effects, e.g., of a sink, without knowing whether the given value originated from a source. CompTaint represents such compositional effects *symbolically* with respect to method parameters. Simple effects like source, sink, and sanitizer amount to unary predicates on symbolic parameters, as well as local and global variables. Input models induce such effects, for example in Code 3 at line 6 a source model for `URL.openStream()` would apply a source effect on its return value. Flow-through effects capture binary dataflow relations among symbolic parameters, e.g., flow from a method parameter to its return value. For example in Code 3 at line 10, a flow model for `InputStream.readAllBytes()` would apply a flow-through effect from its receiver object to its return value. When methods' effects are composed together, i.e., at call sites, the resolution of symbolic effects can trigger combinational logic. For example, when a sink effect of a method parameter is resolved to a call-site argument with a source effect, a source-to-sink flow can be detected; if the first parameter had a sanitize effect instead, the source effect could be removed.

To achieve an adequate level of precision, effects are context-, flow-, field- and object- sensitive. The aforementioned symbolic representation provides context sensitivity, since symbolic values are resolved according to call-site context. We achieve flow sensitivity by computing effects sequentially over program statements and composing effects at call sites in call graph dependency order. To achieve field and object sensitivity, CompTaint follows the modular heap analysis framework of Madhavan et al. [41] and Feng et al. [35], representing effects over object graphs: nodes correspond to objects reachable from parameters, local, and global variables, and edges capture field accesses among objects. In this way, aliasing among accesses is captured by multiple incoming edges to a given node. This representation provides field sensitivity, since distinct fields of any given object may be incident on distinct nodes in the graph, and object sensitivity, since distinct objects in the graph may share the same type.

3.3 Speculative Context Resolution

Next, we discuss the challenge with contextual taint specifications. The fundamental problem stems from two distinct flows, one for the taint, and another for data-flow determining context around the taint. Referring back to the code example in Code 3, the `XStream.fromXML()` method applies a sink effect on `str` only in program contexts where `XStream.allowTypes()` has not been called earlier on its receiver `xs`. Note that when this context dependency is actually resolved, for example at line 14 inside `safeConfigure` method, the tainted value `str` is not available and thus we cannot simply apply a sanitize effect on it. Instead, the validity of the sink effect on `str` at line 11 depends on the state of `xs`. If `safeConfigure` were to be called just before line 11, then this context could be immediately resolved for any context where `deserialize` is called and we could elide the sink effect on `str`. In general however, this context may be resolved inter-procedurally, for example by calling `safeConfigure` before the call to `readUrl` at line 3 when neither the source effect at line 6 nor the sink effect at line 11 have yet manifested. As such, when analyzing `deserialize` method in isolation, the validity of the sink effect at line 11 cannot be resolved since it may indeed be called in a context where `safeConfigure` was never called.

In order to capture such inter-procedural contextual data-flows in our compositional analysis design, we introduce *speculative effects*- an effect that is only valid when additional *context predicates* are also satisfied. A context predicate evaluates a logical combination of primitive predicates on a symbolic method parameter. CompTaint supports two types of predicates that check set membership of the kind(s) of taint or the values of program constants among a specified set of values.

The general support for contextual data-flows in CompTaint necessitated careful handling of speculative effects to handle multiple context predicates, their partial resolution in method summaries, and their interactions with regular or speculative sanitize effects. We elide these details here, but such intricate handling was needed to precisely resolve contextual data-flows in observed real-world code patterns.

```
- target:
    class: XStream
    method: allowTypes
  source: !this       # taints the receiver object
  kind: SAFE_CONFIG   # to capture context

- target:
    class: XStream
    method: fromXML
  sink: !allArgs      # sinks all arguments
  kind: XML_READ
  context: {on:!this , if:{has:NONE, kind:SAFE_CONFIG}}
```

Code 4: CompTaint specifications for handling contextual data-flows in Code 3.

4 OPTIMIZATIONS

This section describes the three optimizations that had a significant impact in CompTaint's deployment.

4.1 Discarding Intermediate Effects

Recall that CompTaint implements a compositional analysis that computes individual method summaries, and analyzes SCC in the method dependency graph to a fix point. This means that we can reduce the peak memory usage by discarding intermediate per-statement effects for previously-analyzed components, loading program components dynamically as they are analyzed, and unloading previously-analyzed program components. CompTaint currently exploits the former, but not the latter two opportunities. Note that within a component, CompTaint must keep the effects for each program statement in order to compute the fixed points of effect iteration sequence limits. Once the fixed points have been computed for a given component, only the method-level effects need be retained, i.e., to apply to call sites; per-statement effects are deallocated. Note that a traditional whole-program analysis would need to keep the state at all program locations in order to reach a fixed point, so this optimization leverages the compositional nature of the analysis.

4.2 Analysis Scope Reduction

Given a set of input specifications S and a call-graph G built globally over all the targets for an instance of the analysis, the goal is to determine parts of the program on which the heavyweight heap-effect analysis can be elided without loss in soundness or precision. The analysis that determines what can be elided must be lightweight.

Soundness Versus Cost At a very high-level, one might start from an insight as follows: a subgraph G' of the whole-program call graph, G, is relevant for the analysis if data-flow from a source of tainted data to a sink occurs in G'. A simple over-approximation of this idea is that if a source and sink are not reachable in a subgraph G' rooted at vertex V over outgoing edge E, then G' could be elided from analysis assuming G' is reachable from roots of G only via V. However, it is straightforward to come up with a counterexample to the above argument.

```
public void entry() {
    valB = foo(valA) // aliases valA and valB
    bar(valA , valB)  // taints valA and sinks valB
}
```

In the example above, we see an invocation to `foo` is followed by invocation to `bar` in method `entry`. While G', the program reachable from `foo` does not taint or sink the data flowing into `foo`, it creates an aliasing relationship between `valA` and `valB`. The subgraph rooted at `bar` then taints `valA` and sinks `valB` creating an insecure data flow. Clearly, eliding G''s analysis will be unsound, however, precisely checking for aliasing will require an analysis as expensive as the full-blown taint analysis.

Eliding Safe Call Graph Roots To avoid analyzing a subgraph it is not sufficient to conclude that the subgraph is devoid of program locations with matching sources or sinks but we need to ascertain that the subgraph does not induce aliasing relations that are then used in the same subgraph or another subgraph in the call graph. Fundamentally, to elide analysis of subgraph G' rooted at V, the analysis needs to consider sources and sinks reachable from V and aliasing created in G'. As a sound over-approximation, we can elide roots R of the call graph from analysis—inferred as *safe roots*—if no matching sources and sinks are reachable $\forall r \in R$, i.e. even if reachable subgraphs from R create aliasing. For example, in the example above, if no source or sink were reachable from the subgraph rooted at the call to `bar`, the root `entry` is safe, hence the entire program reachable from `entry` can be elided from taint analysis.

Adding Precision to Root Elision Given a set of taint specifications S, we derive a set of taint rules T_R. A rule, t in T_R is given by $\{t \mid t \in \ (src_{kind}, sink_{kind})\}$. The single element effects described in § 3 such as source and sink belong to a hierarchy of types called *kinds*. A rule specifies the types of sources and sinks that constitute a vulnerability. For example, a rule to detect XXE vulnerability [16] in Code 3 is specified by source type `UNTRUSTED_DATA_NETWORK` and sink type `XML_READ`. A root of the call graph is only relevant for analysis, if it has reachable source and sink types associated by a rule T_R. The scope-reduction analysis computes all source types and sink types reachable from the roots of the call graph—the entrypoints—and discards the entrypoints which lack any reachable $(src_{kind}, sink_{kind})$ that corresponds to any rule $t \in T_R$. The scope-reduction analysis is lightweight and discards entrypoints that are guaranteed to be safe. The call graph is reused across the scope-reduction analysis and taint analysis. The scope-reduction only requires matching taint specifications—sources and sinks, and propagates matching $(src_{kind}$ and $sink_{kind})$ up to the entrypoints bottom-up in the SCC graph. Note that its analysis domain does not need any notion of access paths or variables. CompTaint uses the results of scope-reduction analysis to recompute a SCC graph using only potentially unsafe entrypoints that are relevant for the analysis; the heavyweight taint analysis that follows uses the reduced SCC graph. In § 6, we discuss the impact of this optimization.

4.3 Caching Invocation Models

CompTaint provides a library of source, sink, and sanitizer specifications that are applied to the program under analysis. Additionally, to model the flow of tainted data in libraries, CompTaint supports *flow models* that apply flow-through effects. These models are applied at invocation sites to different methods, i.e. API calls in the program, and referred to as *invocation models*[2]. Any instantiation of CompTaint must match every model in M, the set of invocation models in the specification library, to every call site. CompTaint as described in § 3 executes a fixed-point iteration on every SCC. In the entire program if CompTaint executes I iterations, and models are matched at every invocation site, say N sites, the time complexity of model matching is $O(M{\times}I{\times}N)$.

In order to avoid repeating a linear scan of all the models every time a call site is analyzed in an iteration, CompTaint creates an index of the target and the models that match the target method(s) at a call site. Once cached, the cost of model matching at a call site is roughly a constant time lookup on the cached models that apply only for the targets at the call site. Asymptotically the time complexity is dominated by I and N, i.e. $O(I{\times}N)$. This is significant since a tool like CompTaint usually has a perpetually growing list of models owing to its vast number of customers and common libraries and SDKs used by its different customers.

The caching described here is an over-approximation and ignores the context resolution described in § 3.3. Further, the scope-reduction analysis and taint analysis equally benefit from caching invocation models. Recall that the scope-reduction analysis only needs caching of source and sink models, unlike taint analysis, which caches sanitizer and flow models in addition. § 6 discusses the impact of this optimization on CompTaint's performance.

5 IMPLEMENTATION

CompTaint is implemented as a modular static data-flow analysis framework for Java. At the heart of this framework lies an abstract reachability algorithms module that traverses over abstract program statements and control-flow edges to compute fixed point. This module can plugin the underlying program representation, and currently we support the Soot Jimple representation [51] for Java bytecode analysis, and the MU Graph representation [25] for Java and Python source code analysis. Before the analysis, we compute the entry-points for the analysis. Entry-points can be annotated explicitly. In addition, we generate a synthetic entry-point for a subject that captures invocations to all public methods in a non-deterministic order. We then build the whole-program call-graph using variable-type analysis (VTA) [50] implementation from Soot Pointer Analysis Research Kit [39] to determine the component dependency order as described in § 3.1. Client analyses extend the reachability analysis by providing implementations for their analysis effects, states and state transformers. CompTaint implements an alias analysis by modeling heap locations as nodes in a graph and program statements with alias effects for assignments, reads and writes inducing edges among them. CompTaint then extends this with the introduction of taint attributes for heap locations and effects of source, sink, sanitize and flow of taint attributes. The aliasing and taint effects are computed and summarized simultaneously for each program component. Throughout the analysis, various relations from program locations to effects on attributes of heap locations are asynchronously written to a tracing database on disk. When CompTaint detects a finding, it uses the tracing database to reconstructs a trace on-demand. In addition to the optimizations discussed in § 4, CompTaint provides a number of options and analysis abstract state size limits for configuring the scope of analysis e.g. state size limiting for SCC components, and making it tractable within various SLAs of its deployment use cases. To ensure we can handle very large inputs where it may not be feasible to terminate, CompTaint has the ability to report partial findings. A trace reconstruction thread runs in parallel and queries the tracing database to report detailed traces as findings are discovered. Note that this works even when we reach the analysis state size budget on an SCC component: due to the compositional nature of the analysis we can just compute an empty summary for the offending component and continue the rest of the analysis.

For security policy enforcement, CompTaint provides an extensible YAML based language to specify rules and models. Rules map interactions of taint and sink kinds to known vulnerabilities. And models specify which API methods induce taint effects of said kinds. CompTaint checks 17 information-flow policy rules to prevent data leaks and top OWASP injection vulnerabilities [15]. It has an extensive library of models for the JDK, Javax, Apache Commons, Guava and popular Java libraries for logging, authentication, serialization and DOM parsing, database connectivity and web-app frameworks.

[2] Note that there are other models in CompTaint that are applied to non-invocations, e.g. *field models* directly capture tainted data-flow through fields and are applied to loads/stores directly.

Additionally, it uses models for the AWS SDK and service APIs for scanning Amazon internal codebases.

Limitations: CompTaint's current implementation is robust for Java bytecode analysis, thoroughly tested for versions 8 and 11 of the JDK. It does not analyze native code and code that uses reflection. It does not currently support runtime dependency injection frameworks. When analyzing concurrent programs, it considers their single-threaded execution, so it does not guarantee detection of data-flows via shared-memory interference, and inter-process communication.

6 EVALUATION

In this section we present experimental results showing the precision and scalability of CompTaint. For evaluating precision we use a labeled dataset consisting of the OWASP Benchmark [14] as well as an internal testsuite. For the performance evaluation, we use a set of open-source Maven Java projects, as well as a set of internal Java Amazon code bases. In both cases, we analyze not only the application packages, but also include the packages in the runtime dependency closure. Note, we do not evaluate precision and recall on this larger dataset because we do not have ground truth labels for this dataset.

6.1 Precision Impact of Contextual Dataflow

To evaluate the precision impact of contextual dataflow models, we use a comprehensive labeled dataset of injection vulnerabilities from the OWASP benchmark [14], an industry standard for evaluating the accuracy and coverage of automated software vulnerability detection tools. Due to the synthetic nature of these benchmarks, we further complement them by adding 120 real-world code examples based on false positives reported by Amazon developers on recommendations reported by different SAST tools on Amazon's internal code reviews. This further includes five additional injection categories not covered by the OWASP tests (shaded bottom five rows in Table 1).

Table 1 summarizes the *false positive rate* (FPR) when running CompTaint on both datasets. On the OWASP benchmarks [14] alone, CompTaint achieves a 100% recall and 13.23% false positive rate on the six applicable categories. The table reports FPR with Baseline and with CompTaint's *contextual data-flow modeling*– i.e. modeling validity of sources, sinks, and sanitizers based on inter-procedural context similar to Code 3. On all injection attack categories, an absence of contextual modeling, causes a precision loss of 9.85% on average, most notably a loss of 50% on code-injection vulnerabilities.

Table 1: False positive rate (FPR) of CompTaint on a labeled dataset of injection vulnerabilities compiled from 1572 OWASP tests, and 120 real-world code examples from the wild with false positives reported by AWS developers on recommendations reported by different SAST tools on code reviews.

	FPR = FP/(FP+TP)	
Vulnerability category	**Baseline**	**CompTaint**
cross-site-scripting	7.61	6.97
ldap-injection	14.71	12.12
os-command-injection	13.46	12.90
path-traversal	13.41	13.41
sql-injection	11.01	10.75
xpath-injection	14.29	10.00
code-injection	50.00	0.00
http-response-splitting	0.00	0.00
log-injection	0.00	0.00
untrusted-deserialization	0.00	0.00
xml-external-entity	50.00	0.00
Average	**15.86**	**6.01**

To evaluate precision on real world code, we use our internal deployment of CompTaint at code-review time. CompTaint posts findings as comments on code reviews, and Amazon developers can mark recommendations as useful, or not useful. Contextual dataflow modeling lowers the false positive rate, computed based on this developer feedback, to less than 20% on internal code [1].

Notably, this significant improvement in precision is achieved with modest effort in writing and maintaining taint specifications. CompTaint uses a library of 1534 taint specifications and only 42 (2.7%) of these require additional contextual modeling.

6.2 Performance Impact of Optimizations

6.2.1 Experimental Setup We provide the methodology for building dependency closures from Maven and internal service repositories.

Maven Analysis Targets We used the *libraries.io* DB [11], which has precomputed dependencies between libraries, and retained Java projects from Maven with Apache, MIT, or BSD-like licenses. This yielded 44,757 projects, not counting versions of the same project. We built the dependency graph modulo versions conservatively counting every version of only the runtime dependencies. We started from the roots, 10,555 projects, and computed the transitive closure of dependencies of each. Since evaluating dependency closures with large overlap is redundant, we reduced overlap as follows. We computed the Jaccard distance between each root and all other roots, based on sets of dependencies, and sorted the candidates by mean Jaccard distance to all others. We selected top 500 projects with latest versions in *libraries.io*. We binned these subject closures on number of .jars, e.g. 1 jar, 2 jars, 3 jars and etc. We limit subject sizes at 20 jars since subjects with 20+ jars timeout on most configurations, making it infeasible to empirically demonstrate the effect of optimizations. We used stratified sampling on this distribution to get 20 closures uniformly distributed across buckets.

Internal Code Analysis Targets We also evaluated our analysis on four large internal applications. We selected code repositories with application code and discarded third-party code, e.g. open-source libraries. For each application, we build the closures from these jars that includes all their bytecode. We include method signatures and type hierarchies for the rest of the classpath. Table 2 shows statistics about the size of these subjects. For the purposes of this evaluation we will use "subject" and "closure" interchangeably, the latter referring to the dependency closure of the former, the root repositories.

6.2.2 Evaluation Questions Our evaluation aims at answering the following questions about the optimizations:

EQ1: *How much effect does scope-reduction analysis have in soundly pruning the size of the analysis problem?* This question should answer how much of the program under analysis is irrelevant given a set of taint specifications.

EQ2: *What's the effect of scope-reduction analysis in reducing analysis time?* This question clarifies if the code elided from analysis is truly expensive to analyze. And does scope-reduction add any overhead to the analysis or is it lightweight in practice?

No.	Maven Subject	#Jar(s)	#Classes	#LOC (Bytecode)
0	jbasics	1	610	67,328
1	etyl-spriter	2	47	14,127
2	simple-servlet-framework	3	219	23,567
3	elaUtils	4	604	127,353
4	lordofthejars-bool	5	369	50,037
5	mind-map-swing-panel	6	327	52,009
6	opentracing-jdbc	7	118	17,599
7	minecloud-core	8	2,132	274,463
8	io7m-jvvfs-shell	9	266	31,869
9	javaflow-maven-plugin	9	524	148,394
10	wagon-gitsite	12	817	164,889
11	wicketstuff-restannotations	12	2,458	285,461
12	trap-js	13	325	41,864
13	radial-encapsulation	14	727	125,073
14	truststore-maven-plugin	15	709	132,242
15	maven-hadoop-plugin	16	770	147,249
16	domdrides-maven-plugin	17	1,493	285,993
17	classycle-maven-plugin	18	953	197,119
18	varnishtest-maven-plugin	19	765	149,319
19	maven-notice-plugin	20	1,567	285,351
No.	Service Subject	#Jar (s)	#Classes	#LOC (Bytecode)
0	anonymous-service-0	34	3,248	542,211
1	anonymous-service-1	4	277	93,204
2	anonymous-service-2	34	4,455	987,345
3	anonymous-service-3	195	6,831	1,133,865

Table 2: Experimental subject closures. Maven subjects include their full transitive closure of runtime dependencies. Service subjects include their closure of internal-code excluding third-party dependencies.

EQ3: *How does model caching improve the time taken by taint analysis and scope-reduction analysis?* We will dive deeper into the effect of model caching on each, its effect on taint analysis, and to analyze if it has an effect on making scope-reduction analysis lightweight.

EQ4: *Does discarding intermediate abstract state impact the total amount of abstract state maintained by the analysis?* We look into total abstract state sizes that can be discarded leveraging compositionality in the lifetime of the analysis.

6.2.3 Experimental results To evaluate the impact of the optimizations, we run CompTaint on Amazon EC2 m5.12xlarge hosts using different configurations as shown below, each with 64 GB Java heap limit and 1 hour time limit.

- **Baseline**: No performance optimizations enabled.
- **ScopeReduction**: Only scope reduction enabled over baseline.
- **ScopeReduction + Caching**: This enables caching invocation models and scope reduction over baseline, i.e. adds caching to ScopeReduction configuration.
- **Discarding**: This enables only discarding of analysis state for already summarized components over baseline.
- **CompTaint**: This enables all analysis optimizations.

It is worth noting that these configurations are analysis semantics preserving and have no effect on the number of detected findings. We confirmed that the number of traces generated from each of the configurations is identical for all subjects.

Impact of Scope Reduction on Analyzed Code In order to answer *EQ1: How much effect does scope-reduction analysis have in soundly pruning the size of the analysis problem?*, we compare Baseline with no performance optimizations and ScopeReduction. Our experiments show that scope-reduction analysis reduces the number of relevant entry points in every subject. The average reduction is 87%, with 89% on Maven and 83% on service code. Figure 1 and Figure 2 show the reduction in the number of entry points, while Figure 3 and Figure 4 show the reduction in the number of methods analyzed. Note that adjudging entry points as safe or irrelevant may not lead to proportionally lower methods analyzed. For example, a large fraction of code may be reachable from a small fraction of relevant entry points. However, in practice, we see substantial reduction in methods analyzed, for the reduced set of entry points above, on average 70%, 72% on Maven and 59% on service closures.

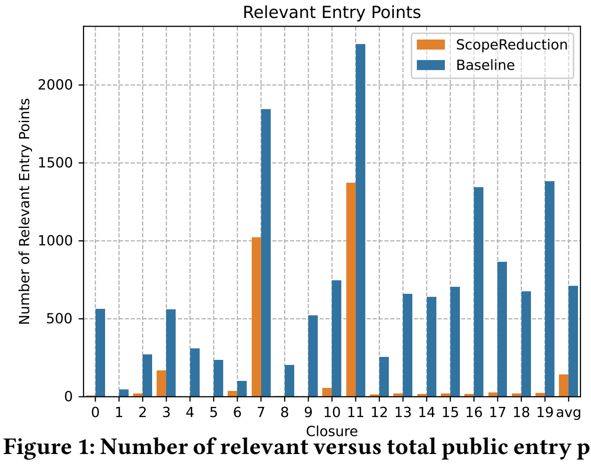

Figure 1: Number of relevant versus total public entry points for Maven closures.

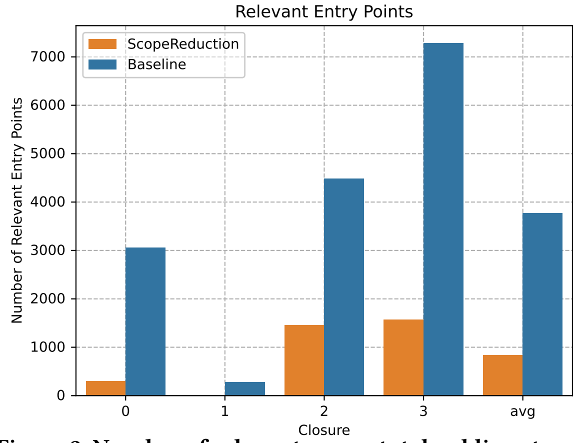

Figure 2: Number of relevant versus total public entry points for service closures.

Impact of Scope Reduction on Analysis Time To address *EQ2: What is the effect of scope-reduction analysis in reducing analysis time?*, we analyze the difference between analysis time with and without scope-reduction analysis, ScopeReduction and Baseline respectively, shown in Figure 5 and Figure 6. Without caching, there is an average 47% reduction for Maven subjects. For service code, there is reduction in analysis time on the 2 subjects, and in fact for the remaining 2, scope-reduction analysis adds overhead to the baseline. Next, in *EQ3*, we discuss how model caching turns this around, and reverts its performance to be a lightweight analysis as hypothesized.

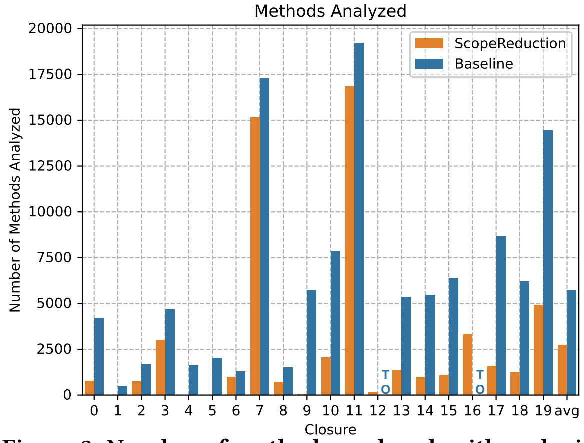

Figure 3: Number of methods analyzed, with and without scope-reduction analysis, for Maven closures. TO stands for timeouts.

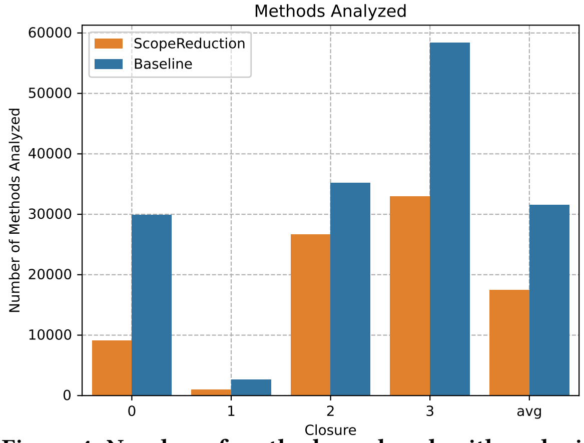

Figure 4: Number of methods analyzed, with and without scope-reduction analysis, for service closures.

Effect of Invocation Model Caching To understand the effect of model caching, we use the configuration called ScopeReduction + Caching. Figure 5 and Figure 6 show the analysis time for ScopeReduction + Caching, used to answer *EQ3: How does model caching improve the time taken by taint analysis and scope-reduction analysis?*. The average time reduction versus baseline rises to 60% for Maven subjects and 19% for service subjects. Hence, combining both optimizations produces worthwhile savings overall. Figure 7 shows the amount of time spent in scope-reduction analysis with and without caching. The average reduction is 89.2%. Note that in several Maven closures we found the analysis time was almost reduced by close to 100 percent, since all entry points were deemed irrelevant by scope-reduction analysis. We do not present these subjects in the figures since they are less interesting, but in practice such scope reduction has proven to be useful in production to reduce time and cost.

On average, the time spent in scope-reduction analysis without caching represents 13.6% of total baseline time. With caching, these percentages drop down to just 1.4% of the total baseline and 7.5% of the total CompTaint time. Overall, 7.5% of analysis time is spent in reducing 70% of code analyzed on average, and significant reduction in overall analysis time as discussed above. We conclude that the cost of the scope-reduction analysis does pay off when combined with model caching.

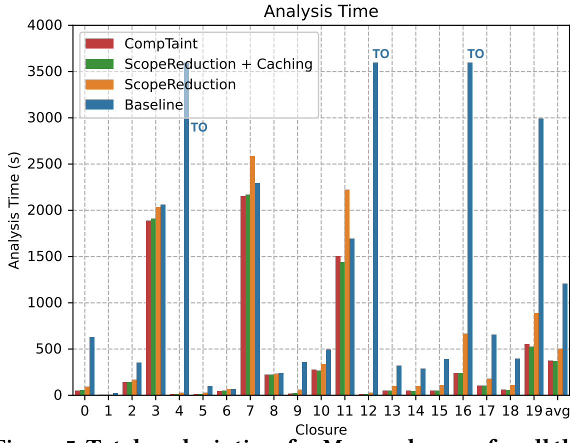

Figure 5: Total analysis time for Maven closures for all the configurations. The label TO adjacent to bars stands for timeouts.

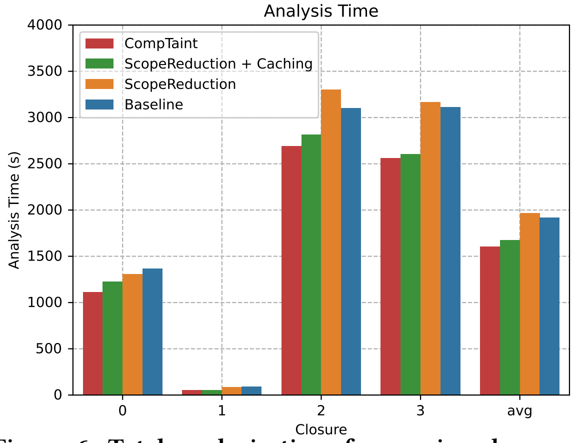

Figure 6: Total analysis time for service closures with different configurations.

Effect of Discarding Abstract State To answer *EQ4: To what extent does discarding intermediate abstract state impact the total amount of abstract state needed to complete the analysis?* we measure the size of the abstract state for Maven closures–nodes and edges in graph modeling the heap, with and without discarding intermediate state. Figure 8 shows the size of the abstract state for Maven closures, with and without discarding intermediate state (minus a few cases where Baseline times out). We observe an average reduction of 94%. We also measured the peak heap memory usage to estimate the effect of this optimization. Although we see reduction in peak heap usage on service code (not shown), peak heap usage depends on the heap budget and frequency and number of garbage collections, and does not always correlate growth in memory usage to increase in analysis problem size.

This dramatic reduction in abstract state size translates to lowering analysis time on some services, e.g. CompTaint versus ScopeReduction + Caching in Figure 6. On Maven, we observe that discarding

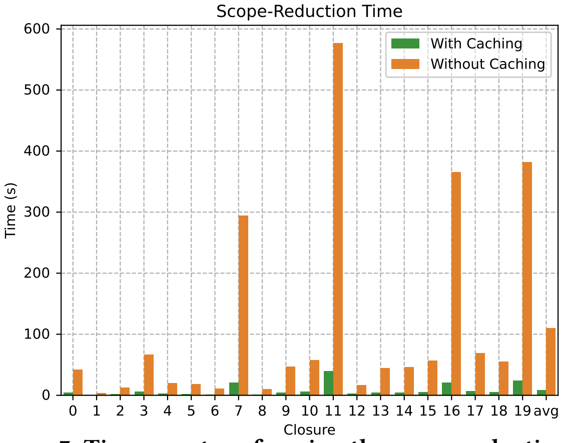

Figure 7: Time spent performing the scope-reduction part of the analysis, for Maven closures.

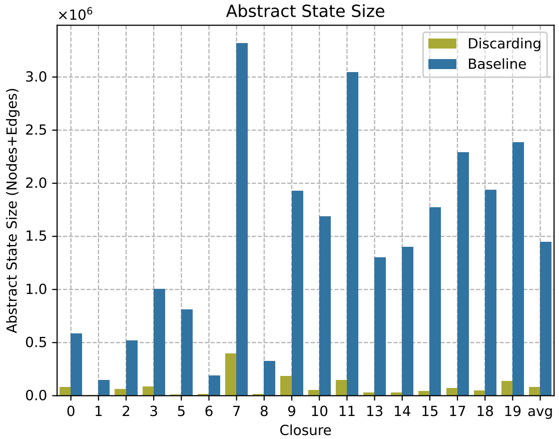

Figure 8: Size of abstract state with and without discarding intermediate state, for Maven closures.

abstract state sometimes come at a small cost in time due to more garbage collections. Nevertheless, holding only necessary state in memory lowers chances of out of memory errors on pathological subjects with complex components that are memory intensive. Overall, CompTaint reduces analysis time over baseline by 69.1% on Maven and by 16.3% on service closures.

7 RELATED WORK

We discuss relevant related work that are geared towards scaling static taint analysis.

RAPID [34] internally uses a IFDS [44] based type-state analysis and *boomerang* based taint analysis [47, 49]. RAPID combines type-state checking and taint analysis to check similar properties as CompTaint. RAPID scales on large subjects only with bounded call-stack depths and cannot reuse analysis results of analyzed components due to context-dependent summarization [28]. RAPID required partitioning [31, 34] in order to scale to subjects of sizes we evaluate at the cost of soundness.

ANTaint [52] is an approach deployed at Alibaba for data leaks detection and data consistency checks. It uses the FlowDroid [27] taint analysis with several changes that improve the precision, recall, and scalability on service-oriented applications (SOAs), such as Spring applications. Another approach tailored to SOAs is JackEE [26], a Doop-based [29] data-flow analysis that demonstrates improvements in precision and scalability. JackEE achieves this via two techniques, a generalized modeling of framework runtime behavior and sound-modulo-analysis model of selected Java data structures. While JackEE shows speed up of 4X compared to other analyses on selected applications, the improvements are tailored to specific frameworks and a subset of standard Java data structures. CompTaint introduces more general optimizations.

P/Taint [36] is another approach based on the Doop framework. In conventional taint analysis approaches, the data-flow analysis is a client of the points-to analysis (e.g., Beacon [38], FlowDroid [27]). The unification of both analyses into a single analysis is the key feature of this approach. P/Taint mainly focuses on improving precision and recall. CompTaint is an industry-scale analysis and emphasizes on maintaining compositionality but like P/Taint unifies taint propagation and heap analysis. Tricoder [45] employs a collection of intraprocedural analyses and uses a microservices architecture for scalability. CompTaint is specifically built for scaling inter-procedurally. Infer and Zoncolan [33] are inter-procedural bi-abduction [30] based analyses that operate at scale in Facebook. A qualitative comparison of the approaches, such as a comparison with CompTaint's compositional contextual modeling, requires further analysis details that are not published to the best of our knowledge. There is a rich body of work on CFL-reachability based static analysis. Graspan [53] models reachability as transitive closure problem on graphs and uses large-scale graph processing for scalability. Grapple extends it to checking finite state properties [54]. CompTaint, combines taint tracking with contextual data-flow modeling, a finite-state property, into a single compositional analysis.

8 CONCLUSION

In this paper we presented an industry-scale compositional static analysis that's deployed internally in Amazon and externally as part of AWS cloud services. We overview the compositional algorithm we implemented and detail our contribution to model contextual data-flow over the heap analysis. We describe the setbacks we experienced before deploying CompTaint in production and how a set of sound optimizations allowed us to productionize the tool. We measure the precision benefit of contextual data-flow modeling. We systematically built benchmarks to demonstrate challenges in real deployment scenarios that require analyzing large artifacts, and present the effect of the optimizations on the subjects.

References

[1] [n.d.]. Amazon CodeGuru. https://aws.amazon.com/codeguru.

[2] [n.d.]. Amazon CodeGuru Reviewer. https://docs.aws.amazon.com/codeguru/latest/reviewer-ug/welcome.html.

[3] [n.d.]. Amazon Elastic Container Registry (Amazon ECR). https://aws.amazon.com/ecr/.

[4] [n.d.]. Amazon Inspector. https://https://aws.amazon.com/inspector/.

[5] [n.d.]. AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

[6] [n.d.]. CodeQL. https://codeql.github.com/.

[7] [n.d.]. CodeQL Github Action. https://github.com/github/codeql-action.

[8] [n.d.]. Cross Site Scripting (XSS). https://owasp.org/www-community/attacks/xss/.

[9] [n.d.]. Docker Hub. https://hub.docker.com/.

[10] [n.d.]. Identity Theft Resource Centre, 2022 Data Breach Report. https://www.idtheftcenter.org/publication/2022-data-breach-report/.

[11] [n.d.]. Libraries.io. https://libraries.io/.

[12] [n.d.]. Maven Central Repository. https://maven.apache.org/.

[13] [n.d.]. Optus Data Breach. https://www.cshub.com/attacks/news/iotw-everything-we-know-about-the-optus-data-breach.

[14] [n.d.]. OWASP Benchmark: The OWASP Benchmark Project. https://owasp.org/www-project-benchmark.

[15] [n.d.]. OWASP Top Ten 2017: Injection. https://owasp.org/www-project-top-ten/2017/A1_2017-Injection.

[16] [n.d.]. OWASP XXE: OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html.

[17] [n.d.]. Plume Taint Specification. https://github.com/plume-oss/benchmarking/blob/main/experiments/src/main/resources/taint_definitions.yaml.

[18] [n.d.]. Scanning AWS Lambda Functions with Amazon Inspector. https://docs.aws.amazon.com/inspector/latest/user/scanning-lambda.html.

[19] [n.d.]. Snyk Container. https://snyk.io/product/container-vulnerability-management/.

[20] [n.d.]. SQL Injection. https://owasp.org/www-community/attacks/SQL_Injection.

[21] [n.d.]. Twitter Accounts Data Breach. https://www.cshub.com/attacks/news/54-million-twitter-accounts-reportedly-on-sale-in-hacking-forum.

[22] [n.d.]. XPath Injection. https://owasp.org/www-community/attacks/XPATH_Injection.

[23] [n.d.]. Xstream API Documentation. https://x-stream.github.io/javadoc/index.html.

[24] [n.d.]. Xstream Security Aspects. https://x-stream.github.io/security.html.

[25] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* 45, 12 (2019), 1170–1188. https://doi.org/10.1109/TSE.2018.2827384

[26] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static Analysis of Java Enterprise Applications: Frameworks and Caches, the Elephants in the Room *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 794–807. https://doi.org/10.1145/3385412.3386026

[27] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[28] Eric Bodden. 2018. The Secret Sauce in Efficient and Precise Static Analysis: The Beauty of Distributive, Summary-Based Static Analyses (and How to Master Them). In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 85–93. https://doi.org/10.1145/3236454.3236500

[29] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[30] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9058)*, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

[31] Maria Christakis, Thomas Cottenier, Antonio Filieri, Linghui Luo, Muhammad Numair Mansur, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. 2022. Input splitting for cloud-based static application security testing platforms. In *ESEC/FSE 2022*. https://www.amazon.science/publications/input-splitting-for-cloud-based-static-application-security-testing-platforms

[32] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[33] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

[34] Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. 2021. Rapid: Checking API usage for the cloud in the cloud. In *ESEC/FSE 2021*. https://www.amazon.science/publications/rapid-checking-api-usage-for-the-cloud-in-the-cloud

[35] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9458)*, Xinyu Feng and Sungwoo Park (Eds.). Springer, 465–484. https://doi.org/10.1007/978-3-319-26529-2_25

[36] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified Points-to and Taint Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 102 (oct 2017), 28 pages. https://doi.org/10.1145/3133926

[37] Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2, Article 29 (jun 2016), 47 pages. https://doi.org/10.1145/2931098

[38] Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-chieh Shan. 2012. An Analysis of the Mozilla Jetpack Extension Framework *(ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 333–355. https://doi.org/10.1007/978-3-642-31057-7_16

[39] Ondrej Lhoták and Laurie J. Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2622)*, Görel Hedin (Ed.). Springer, 153–169. https://doi.org/10.1007/3-540-36579-6_12

[40] Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2019. Rethinking Incremental and Parallel Pointer Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 1 (2019), 1–31.

[41] Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. 2015. A Framework For Efficient Modular Heap Analysis. *Found. Trends Program. Lang.* 1, 4 (2015), 269–381. https://doi.org/10.1561/2500000020

[42] Rajdeep Mukherjee, Omer Tripp, Ben Liblit, and Michael Wilson. 2022. Static analysis for AWS best practices in Python code. In *ECOOP 2022*. https://www.amazon.science/publications/static-analysis-for-aws-best-practices-in-python-code

[43] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *NDSS*, Vol. 14. 1125.

[44] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. ACM, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[45] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 598–608.

[46] Johannes Späth, Karim Ali, and Eric Bodden. 2017. IDE^{al}: efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 99:1–99:27. https://doi.org/10.1145/3133923

[47] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-Sensitive Data-Flow Analysis Using Synchronized Pushdown Systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290361

[48] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-sensitive Data-flow Analysis Using Synchronized Pushdown Systems. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages* 3, POPL, Article 48 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290361

[49] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *Proceedings of the 30th European Conference on Object-Oriented Programming*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

[50] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical virtual method call resolution for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 2000, Minneapolis, Minnesota, USA, October 15-19, 2000*, Mary Beth Rosson and Doug

Lea (Eds.). ACM, 264–280. https://doi.org/10.1145/353171.353189

[51] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?. In *Proceedings of the 9th International Conference on Compiler Construction (CC '00)*. Springer-Verlag, Berlin, Heidelberg, 18–34.

[52] Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. 2020. Scaling Static Taint Analysis to Industrial SOA Applications: A Case Study at Alibaba. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1477–1486. https://doi.org/10.1145/3368089.3417059

[53] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-Machine Disk-Based Graph System for Interprocedural Static Analyses of Large-Scale Systems Code *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 389–404. https://doi.org/10.1145/3037697.3037744

[54] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. 2019. Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale Systems Code. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 38, 17 pages. https://doi.org/10.1145/3302424.3303972

Received 2023-05-18; accepted 2023-07-31