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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ESEC/FSE ’23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
https://doi.org/10.1145/3611643.3613889

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

ACM Reference Format:

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 re-analysis 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 compositionalality, 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 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.

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% frag. 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

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

Code 1: Cipher sanitizer for sensitive data leak.

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

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

```java
1 parse ( String url ) {
2 XStream xs = new XStream ();
3 readUrl ( url , xs );
4 }
5 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 safeConfigure ( XStream xs ) {
14 xs . allowTypes ( new Class [ ] { Snapshot . class ,
15 Envelope . class });
16 }
```

Code 3: XML external entity sink.

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

Using allowsTypes. 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 str and not the tainted variable str, and the sink fromXML () is neutralized due to safe state of the str object. This example also illustrates that precisely tracking the context requires an inter-procedural analysis.

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
We observed that 19.5% out of 82 Maven dependency closures timed
Taint
(a) Given a set of taint specifications \( S \) (see § 6.2.1) before deploying
fixed points by applying abstractions to approximate effects [32].

iteration sequences to ensure methods’ effects capture all possible
of its statements. Since the call graph may be cyclic, due to either
recursion or call-graph imprecision. To obtain the desired depen-
dency 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, Comp-
Taint computes call graphs via the variable type analysis (VTA) algo-
rithm [50] implemented by SPARK [39]. This algorithm utilizes an in-
expensive 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 poten-
tial targets. We achieve this in linear time by computing and prop-
agating types over the strongly connected components of the PAG.

3 COMPOSITIONAL ANALYSIS

In this section we give a high level overview of CompTaint’s com-
positional analysis algorithm. To handle heap aliasing composi-
tionally, 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, Comp-
Taint computes its effects using the effects computed for the methods
it calls. § 3.1 describes the definition of a component in presence of re-
cursion. 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 depend-
cy 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 po-
tentially invoked, i.e., call targets. We obtain a dependency graph
among methods by identifying call sites with their enclosing meth-
ods. However, this dependency graph may be cyclic, due to either
recursion or call-graph imprecision. To obtain the desired depen-
dency 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.

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 spec-
fications 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 param-
eters. Simple effects like source, sink, and sanitizer amount to unary
predicates on symbolic parameters, as well as local and global vari-
ables. 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 rela-
tions among symbolic parameters, e.g., flow from a method paramete-
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 read() 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 necessitates 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.

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.

```java
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.
Adding Precision to Root Elision

For example, a rule to detect XXE vulnerability \[16\] in applied to loads/stores directly.

Field models in the specification library, to every call site. CompTaint provides a library of source, sink, and sanitizer specifications—sources and sinks, and propagates matching \((src_{kind}, sink_{kind})\) 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 recomputes 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.

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 \(\{t | t \in \{src_{kind}, sink_{kind}\}\}\). 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 \(\text{UNTRUSTED\_DATA\_NETWORK}\) and sink type \(\text{XML\_READ}\). 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.

CompTaint uses the tracing database to reconstructs a trace on-demand. 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. \(O(1 \times N)\). 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.

Note that the analysis domain does not need any notion of access paths or variables. CompTaint uses the results of scope-reduction analysis to recomputes 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 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

\[N\] sites, the time complexity of model matching is \(O(M \times I \times N)\).

5 IMPLEMENTATION

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. \(O(1 \times N)\). 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.

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—inflected 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 \(\text{bar}\), the root entry is safe, hence the entire program reachable from entry can be elided from taint analysis.
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 tainted and sink kinds to known vulnerabilities. And models specify which API methods induce taint effects of said kinds. CompTaint checks 17 information-flow rules to prevent data leaks and top OWASP injection vulnerabilities [15]. It has an extensive library of models for the JDK, Jaxva, Apache Commons, Guava and popular Java libraries for logging, authentication, serialization and DOM parsing, database connectivity and web-app frameworks. 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 testset. 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 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.

<table>
<thead>
<tr>
<th>Vulnerability category</th>
<th>FPR - FP/(FP+TP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cross-site-scripting</td>
<td>7.61</td>
</tr>
<tr>
<td>ldap-injection</td>
<td>14.71</td>
</tr>
<tr>
<td>os-command-injection</td>
<td>13.46</td>
</tr>
<tr>
<td>path-traversal</td>
<td>13.41</td>
</tr>
<tr>
<td>xpquery-injection</td>
<td>14.29</td>
</tr>
<tr>
<td>code-injection</td>
<td>50.00</td>
</tr>
<tr>
<td>http-response-splitting</td>
<td>0.00</td>
</tr>
<tr>
<td>log-injection</td>
<td>0.00</td>
</tr>
<tr>
<td>untrusted-external-entity</td>
<td>0.00</td>
</tr>
<tr>
<td>average</td>
<td>15.86</td>
</tr>
</tbody>
</table>

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.

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.

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.
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.

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?

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.

### Table 2: Relevant Entry Points

<table>
<thead>
<tr>
<th>No.</th>
<th>Maven Subject</th>
<th>#Jar(s)</th>
<th>#Classes</th>
<th>#LOC (Bytecode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>phaxs</td>
<td>1</td>
<td>670</td>
<td>67382</td>
</tr>
<tr>
<td>2</td>
<td>etyl-spriter</td>
<td>2</td>
<td>47</td>
<td>14127</td>
</tr>
<tr>
<td>3</td>
<td>simple-servlet-framework</td>
<td>3</td>
<td>219</td>
<td>23567</td>
</tr>
<tr>
<td>4</td>
<td>elcats</td>
<td>4</td>
<td>604</td>
<td>127355</td>
</tr>
<tr>
<td>5</td>
<td>lordofthejars-boot</td>
<td>5</td>
<td>369</td>
<td>50037</td>
</tr>
<tr>
<td>6</td>
<td>mind-map-swing-panel</td>
<td>6</td>
<td>327</td>
<td>52009</td>
</tr>
<tr>
<td>7</td>
<td>opentracing-jdbc</td>
<td>7</td>
<td>118</td>
<td>17599</td>
</tr>
<tr>
<td>8</td>
<td>maven-core</td>
<td>8</td>
<td>2132</td>
<td>274465</td>
</tr>
<tr>
<td>9</td>
<td>io7m-jvfs-shell</td>
<td>9</td>
<td>266</td>
<td>31869</td>
</tr>
<tr>
<td>10</td>
<td>javafflow-maven-plugin</td>
<td>9</td>
<td>524</td>
<td>148394</td>
</tr>
<tr>
<td>11</td>
<td>wagon-gitsite</td>
<td>12</td>
<td>817</td>
<td>164889</td>
</tr>
<tr>
<td>12</td>
<td>wicketstuff-restannotations</td>
<td>12</td>
<td>2458</td>
<td>285461</td>
</tr>
<tr>
<td>13</td>
<td>trap-js</td>
<td>13</td>
<td>325</td>
<td>41864</td>
</tr>
<tr>
<td>14</td>
<td>radial-encapsulation</td>
<td>14</td>
<td>727</td>
<td>125073</td>
</tr>
<tr>
<td>15</td>
<td>truststore-maven-plugin</td>
<td>15</td>
<td>709</td>
<td>132242</td>
</tr>
<tr>
<td>16</td>
<td>maven-hadoop-plugin</td>
<td>16</td>
<td>770</td>
<td>147249</td>
</tr>
<tr>
<td>17</td>
<td>domdrides-maven-plugin</td>
<td>17</td>
<td>1493</td>
<td>283993</td>
</tr>
<tr>
<td>18</td>
<td>classycle-maven-plugin</td>
<td>18</td>
<td>953</td>
<td>197119</td>
</tr>
<tr>
<td>19</td>
<td>varnish-test-maven-plugin</td>
<td>19</td>
<td>765</td>
<td>149319</td>
</tr>
<tr>
<td>20</td>
<td>maven-restice-plugin</td>
<td>20</td>
<td>1367</td>
<td>285351</td>
</tr>
</tbody>
</table>

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.

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.
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.

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.

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 timeouts.

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.

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

7 RELATED WORK

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

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.

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 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 an IFDS [44] based type-state analysis and boomerang based 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 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 intra-procedural 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. Grasp [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

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


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