Practical Type-Based Taint Checking and Inference

Nima Karimipour
University of California, Riverside, CA, USA

Kanak Das
University of California, Riverside, CA, USA

Manu Sridharan
University of California, Riverside, CA, USA

Behnaz Hassanshahi
Oracle Labs, Brisbane, Australia

Abstract

Many important security properties can be formulated in terms of flows of tainted data, and improved taint analysis tools to prevent such flows are of critical need. Most existing taint analyses use whole-program static analysis, leading to scalability challenges. Type-based checking is a promising alternative, as it enables modular and incremental checking for fast performance. However, type-based approaches have not been widely adopted in practice, due to challenges with false positives and annotating existing codebases. In this paper, we present a new approach to type-based checking of taint properties that addresses these challenges, based on two key techniques. First, we present a new type-based tainting checker with significantly reduced false positives, via more practical handling of third-party libraries and other language constructs. Second, we present a novel technique to automatically infer tainting type qualifiers for existing code. Our technique supports inference of generic type argument annotations, crucial for tainting properties. We implemented our techniques in a tool TaintTyper and evaluated it on real-world benchmarks. TaintTyper exceeds the recall of a state-of-the-art whole-program taint analyzer, with comparable precision, and 2.93X–22.9X faster checking time. Further, TaintTyper infers annotations comparable to those written by hand, suitable for insertion into source code. TaintTyper is a promising new approach to efficient and practical taint checking.

2012 ACM Subject Classification Software and its engineering → Software verification and validation; Security and privacy → Software security engineering

Keywords and phrases Static analysis, Taint Analysis, Pluggable type systems, Security, Inference

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.18

Related Version *Extended Version*: `https://arxiv.org/abs/2504.18529` [29]

Supplementary Material *Software*: `https://github.com/ucr-riple/TaintTyper`
archived at `swh:1:dir:18769875fdd197811ba90f1008f98cc3d8c47fe0`
Software (ECOOP 2025 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.11.2.7`

Funding This research was supported in part by the National Science Foundation under grants CCF-2007024, CCF-2223826, and CCF-2312263, and a gift from Oracle Labs.

1 Introduction

Software security is of critical importance to society, as security vulnerabilities can have severe financial and safety impacts. Many security vulnerabilities can be described as an undesirable flow of *tainted* data. For example, program inputs possibly controlled by an attacker are tainted, and should not flow to sensitive program operations without proper sanitization. *Static taint analysis* aims to automatically discover these dangerous data flows, and many approaches to static taint analysis have been proposed [13, 44, 11, 27, 24, 4, 21, 45, 34, 46, 26].

We desire a static taint analysis with the following properties:

- *high precision*, i.e., few false positive reports;
- *high recall*, i.e., few missed vulnerabilities;
- *fast running times*, enabling checking during continuous integration (CI) or even on local builds; and
- *applicability* to existing code bases.

Previous approaches do not satisfy all of these properties. Most existing tools perform whole-program analysis, using inter-procedural alias and dataflow analysis to discover tainted flows. Such approaches are applicable to extant code and can have high precision and recall, but they are difficult to scale to large programs, which can have millions of lines of code [13, 44]. Such analyses can take hours to complete, making them insuitable for CI deployment or local developer runs. Incremental analysis techniques can speed re-analyzing code after small changes [13, 15, 42], but they may miss issues [41] or still consume excessive resources [42].

An alternative approach is type-based checking using type qualifiers [23], as embodied in pluggable type checkers [36, 19]. A type-based approach performs *modular* checking for tainted flows, combining intra-procedural analysis with annotations at method boundaries to capture inter-procedural flows. The Checker Framework [36, 19] includes a type-based tainting checker [2]. Such approaches can have high recall, and they have fast running times, as the use of annotations *completely obviates* the need for inter-procedural analysis. They are also naturally incremental due to modularity, yielding an automatic and significant speedup (nearly 10X in our initial measurements) for small code changes with modern build systems [6]. However, extant type-based approaches lack both precision and applicability. For precision, the previous checker [2] treats third-party libraries and various language constructs too conservatively, leading to excessive false positives. For applicability, running a type-based approach on extant code requires adding annotations, a significant up-front effort that limits adoption.

In this paper, we present a new approach to type-based checking for taint properties that achieves all of our desired properties: high precision and recall, fast running times, and applicability. In contrast to the extant approach [2], we design our checker to prioritize usability and flexibility over full soundness. We handle un-annotated library methods as *polymorphic* by default [27], i.e., only returning tainted data when it is passed in, dramatically reducing the need to annotate such methods. We also introduce more practical handling of a variety of language constructs to reduce false positives. While these techniques can in theory introduce unsoundness, our extensive evaluation showed that the impact on analysis recall was minimal.

We also present a novel technique to automatically infer tainting type qualifiers, achieving applicability to existing code. Our technique extends a recent search-based approach for inferring nullability annotations [30] with several important new features. We develop a novel algorithm for inferring type annotations on generic type arguments, crucial for tainting; this problem was left open in recent work [30, 31]. Our algorithm can also infer polymorphic method annotations when generic types are not present. We introduce an optimized handling of local variable annotations to further improve inference performance.

We implemented our technique in a tool TaintTyper and evaluated it on a range of benchmarks. TaintTyper was able to infer annotations comparable to human-written annotations and suitable for insertion into source code. Further, once inference was completed, TaintTyper had higher recall than a state-of-the-art taint analyzer on real-world benchmarks, with comparable precision, and 2.93X–22.9X faster checking time. Hence, the evaluation showed that TaintTyper achieved the precision, recall, speed, and applicability goals outlined above. An ablation experiment showed that our new checker and inference features were crucial for TaintTyper's effectiveness.

This paper makes the following key contributions:

- We present a new, more practical type-based tainting checker, with improved handling of third-party libraries and other language constructs.
- We describe a novel inference technique for tainting types. Crucially, the technique handles generic type arguments, and includes an important new optimization to reduce inference time.
- We present a detailed experimental evaluation showing our approach makes type-based taint checking much more practical for real-world projects, significantly improving over the state-of-the-art.

The implementation of TaintTyper is available at `https://github.com/ucr-riple/TaintTyper`.

2 Background

In this section, we provide background on typical approaches to taint analysis (Sec. 2.1) and the type-based approach (Sec. 2.2), comparing them and motivating our techniques.

2.1 Taint Analysis

Taint analysis aims to discover *information flow vulnerabilities* in code [18], an undesirable flow of information from some designated *source* of values to a designated *sink* operation. Our attacker model assumes that the attacker controls inputs originating from untrusted sources and seeks to reach sensitive sinks. In this context, sources include typical mechanisms that receive data from external origins, such as reading from the network, accessing files, or handling user input. Conversely, sinks are operations that have security-critical effects, such as writing to a file, sending data over the network, or executing system or database commands. E.g., for an SQL injection attack, the source is an input from a potential attacker and the sink is a database query, while for a cross-site scripting (XSS) attack, the sink renders output to a web page. Information flow may be *direct*, solely involving data dependencies, or *indirect*, potentially involving control dependence. As with most practical static taint analyses, we consider only direct information flow in this work.

The most common approach to static taint analysis is to perform an inter-procedural (whole-program) dataflow analysis to discover source-to-sink data flows [13, 44, 11, 27, 24, 4, 21, 45, 34, 46, 26]. For languages like Java, inter-procedural analysis requires computing a call graph, typically using whole-program pointer analysis [40]. Tracking of tainted data flows also requires handling of pointer aliasing, to account for flows through object fields and data structures like lists. Precise reasoning about call graphs and pointer aliasing is the key scalability bottleneck for whole-program taint analysis. For example, a recent approach [25] required over 3 hours and 153GB of RAM to precisely analyze the full Java standard library, and real-world programs may grow much larger. Approaches to make such analyses incremental have been proposed [13, 15, 42], which could speed up re-analysis of a code base after a small change. But, such approaches can miss issues [41] or still require significant time or memory [42].

Tainting rules capturing which source-sink flows may be vulnerable are typically provided as input to a taint analysis tool. Such rules may also capture information about *validator* and *sanitizer* operations, whose use may make a source-sink flow safe. Discovering tainting rules is itself non-trivial and has been a subject of much research [38, 17, 20, 33]. In this work, we focus on the core analysis problems and assume tainting rules have been provided; any improvements in taint rule discovery could be easily combined with our approach.

2.2 Type-Based Taint Analysis

2.2.1 Basics

Type-based taint analysis is built on the ideas of pluggable type systems [36, 19]. A pluggable type system defines a set of *type qualifiers* that refine the built-in types of a language, further restricting the kind of value an expression may evaluate to. We write type qualifiers by prefixing with an `@` character, using Java annotation syntax. For tainting, the main qualifiers are `@Tainted`, for expressions that *may* be influenced by (data-dependent on) a source, and `@Untainted`, for expressions that *must not* be influenced by a source. Tainting rules are provided in the form of `@Tainted` qualifiers on source operations and `@Untainted` qualifiers on sink operations.

A pluggable type system must also define a subtyping relationship $\leq$ between qualifiers. For tainting, we have `@Untainted` $T \leq$ `@Tainted` T for any base type T; `@Untainted` values may safely flow to (possibly) `@Tainted` locations, but not vice-versa. Enforcing this property requires checking for subtype compatibility at all pseudo-assignments in the program, including assignments to variables / fields, parameter passing, and returns. To reduce the number of explicit annotations required, pluggable type systems interpret unqualified types as having a default qualifier. For tainting, the default qualifier is `@Tainted`. The following example illustrates this checking:

```
class Ex {
  @Tainted String f1; @Untainted String f2;
  void m1(@Untainted String s) {
    f1 = s; // no error: @Untainted assigned to @Tainted
  }
  void m2(@Tainted String t) {
    m1(t); // error: @Tainted passed to @Untainted
    f2 = t; // error: @Tainted assigned to @Untainted
  }
}
```

In the code, `@Tainted String f1` could have been written as just `String f1` due to defaulting. Local variables are not subject to defaulting: their types are inferred using flow-sensitive dataflow analysis [36, 19], which also enables handling of taint validators. By default, the Checker Framework's Tainting Checker [2] treats all code, including unannotated third-party libraries, with the same defaulting rules. So, since `@Tainted` is the default qualifier, any value returned by a third-party method is assumed to be tainted. This approach leads to too many false positives in practice; Sec. 4.1 describes our more practical handling of such code.

2.2.2 Method calls

Pluggable type systems must also check that method overriding respects the subtyping rules for the type qualifiers. Return types must be covariant in overriding methods, and parameter types must be contravariant. Consider the following (erroneous) example:

```
class Super {
  @Untainted String foo() { return "hi"; }
}
class Sub extends Super {
```

```
  // invalid override!
  @Tainted String foo() { return source(); }
}
```

Sub.foo() cannot be allowed to return a @Tainted String, since code written to handle Super objects may expect foo() to return an @Untainted String. Without this checking, the following vulnerable code would pass the checker:

```
void process(Super s) { sink(s.foo()); }
process(new Sub());
```

Given this method override checking, method calls can be handled in the pluggable types approach *without computing a call graph*. At a call c to a method with declared target P.m, the checker needs only to check that the parameters passed and return value use at c are consistent with the type of P.m itself. If at runtime, an overriding method Q.m is invoked at c, override checking ensures that the tainting behavior of Q.m is consistent with P.m, so no vulnerability is possible. This ability to check calls without a call graph yields a huge scalability benefit for the pluggable types approach.

2.2.3 Data structures / polymorphism

Pluggable type systems use parametric polymorphism to support storing differing types of data in different instances of a data structure. Consider the following example:

```
List<@Tainted String> taintedStrs = new ArrayList<>();
List<@Untainted String> untaintedStrs = new ArrayList<>();
taintedStrs.add(source()); untaintedStrs.add("safe");
sink(taintedStrs.get(0)); // error reported
sink(untaintedStrs.get(0)); // no error reported
```

Here we have two ArrayList instances, one holding possibly-tainted strings and the other holding untainted strings. The taintedness of list contents is captured by adding a taint qualifier to the generic type argument, e.g., List<@Tainted String>. With these type declarations, the pluggable type checker reports an error at the first sink call while also proving that the second sink call is safe.

To achieve similar precision, a whole-program static analysis must use *context sensitivity*: each call to the relevant ArrayList methods must be analyzed separately, and ArrayList's internal state must be represented with a context-sensitive heap abstraction [40]. Context sensitivity significantly increases the running time of such analyses. The pluggable types approach uses qualified type arguments to achieve similar precision with much less cost.

@PolyTaint annotations can be used for polymorphic methods that do not already use type variables. Consider, e.g., the parentPath function in Fig. 1. Here, the return value of a call to parentPath should only be considered @Tainted if the parameter for that call is @Tainted; the @PolyTaint annotations capture this property. Again, a standard whole-program analysis can achieve this precision with context sensitivity, but at greater cost to scalability.

It is possible that different objects of a non-generic class vary in terms of whether tainted data is stored in their fields. Such cases can be handled in whole-program analysis using field sensitivity, which models each field of each (abstract) object separately. Typically, no such modeling is used in the pluggable types approach, outside of generic types. We have found that this rarely leads to false positives; qualified generic type arguments and @PolyTaint nearly always suffice for good precision in practice.

```
1 Map<String, +@Untainted String> paths = ...;
2 +@PolyTaint String parentPath(+@PolyTaint String path) {
3   return path.substring(0, path.lastIndexOf("/"));
4 }
5 void exec(String name) {
6   String path = paths.get(name);
7   String parent = parentPath(path);
8   sink(Paths.get(parent).toAbsolutePath().toFile());
9 }
10 void sink(@Untainted String t) { ... }
```

Figure 1 Motivating example for inference. Green text indicates where annotations are inserted by TaintTyper.

2.2.4 Benefits

A typechecking approach to tainting scales well since the checking is fully modular: each method can be checked in isolation given only the type signatures of fields it accesses and methods it invokes. Checking is also incremental: after a change, only the code that needs to be re-compiled needs to be re-checked. Beyond improved scalability, a type-based approach to taint checking has various other advantages [36, 19]. Qualifiers serve as machine-checked documentation of tainting properties and invariants, and can also aid in safely performing refactorings. And, errors from the type-based approach can be more understandable, since they only require reasoning about local code and the types of related functions, not an inter-procedural trace. Because our analysis is type-based, it operates over type annotations rather than runtime behavior, meaning the attacker's knowledge of the program's internals does not fundamentally affect the analysis. We assume the attacker cannot modify the program's source code.

Currently, adopting type-based taint checking imposes a significant burden on programmers, as they must manually annotate their own code and any relevant third-party code. Our checking and inference techniques significantly reduce this burden, making the type-based approach more practical.

3 Motivating Example and Approach

Example. Fig. 1 gives a motivating example for our approach. (Disregard the green inferred annotations for the moment.) The `sink` method (line 10) requires an `@Untainted` argument. Assuming the values in the `paths` map cannot be tainted, the `sink` call at line 8 is safe. We shall show how the enhanced checker and inference of TaintTyper can automatically annotate and validate this example.

The first challenge is the use of library methods like `Paths.get`, `toAbsolutePath`, and `toFile` on line 8. Without manual annotation, the previous type-based approach [2] treats these calls as returning a `@Tainted` value by default. Our approach treats most unannotated methods as polymorphic [27]: it assumes they only return `@Tainted` data when a parameter is `@Tainted`. Therefore, by default, TaintTyper determines that the expression passed to `sink` can only be `@Tainted` if the `parent` variable is `@Tainted`, a correct handling for this example.

The `parent` variable is assigned the result of calling `parentPath(path)` (line 7), where `path` is a value in the `paths` map (line 6). As noted above, the values in the `paths` map are not tainted, which is captured by giving `paths` the type `Map<String,@Untainted String>` (line 1); this type makes the `path` variable `@Untainted` at line 6. The `parentPath` function

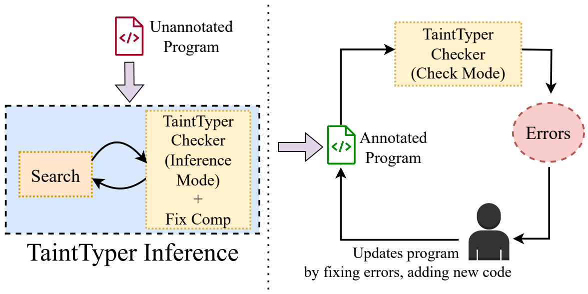

Figure 2 High-level architecture of TaintTyper.

can only return `@Tainted` data if its argument is `@Tainted`, captured with `@PolyTaint` annotations (line 2). With these annotations, the TaintTyper checker reports no warning for this code, as desired.

The example shows that to be effective, TaintTyper must support inferring annotations on generic type arguments and `@PolyTaint` annotations. The generic type argument support is required to eliminate the error, and `@PolyTaint` properly captures the behavior of the `parentPath` method.[1] The inference technique of Sec. 5 can successfully infer the necessary green annotations in Fig. 1.

Our approach. Fig. 2 shows the high-level architecture of TaintTyper. Given an unannotated program, TaintTyper first performs inference to create an annotated version of the program. The inference improves on a recent search-based technique [30] that repeatedly runs a checker to determine the best set of annotations to insert. Here, the checker is our new type-based taint checker, combined with a new fix computation algorithm to enable inference of type argument annotations and `@PolyTaint`. The inference step only needs to run once. Afterward, developers need only run the TaintTyper checker. They can fix the errors initially reported by the checker and also write new code (with annotations) that will be verified by the checker. Since the checker is type based, it runs much faster than a whole-program static analysis, enabling quick turnaround times, less CI resource usage, and even checking on local builds.

4 Practical Type-Based Checking

This section details the new features in TaintTyper's type checker that reduce false positives in real-world code, making the checker practical. We discuss handling of unannotated code, and then new handling of other language constructs.

4.1 Unannotated Code

TaintTyper specially handles interactions with unannotated, unchecked code, typically written by a third-party. Given the prevalence of third-party libraries lacking taint annotations in modern Java programs, such interactions occur commonly. By default, the Checker Framework's Tainting Checker [2] uses `@Tainted` annotation for all code, whether from source or from libraries. So, all unannotated library methods are assumed to have a `@Tainted` return type, yielding too many false positives to be usable.

[1] For this code excerpt, the parameter and return of `parentPath` could be marked `@Untainted`, but this would unnecessarily force all other callers of `parentPath` to pass in `@Untainted` data.

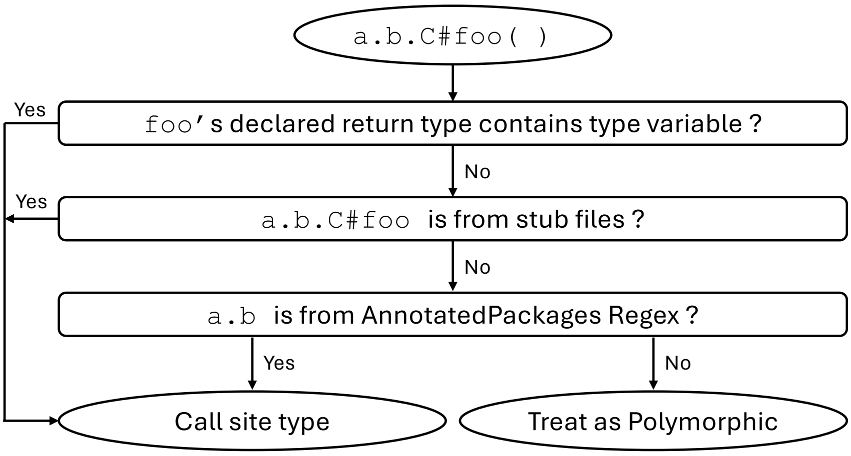

Figure 3 Logic for determining the return type qualifier for a method call, accounting for unannotated code.

TaintTyper adopts a *polymorphic by default* handling of calls to unannotated methods, extending a technique from previous work [27]. This approach treats the return type and all parameter types (including the receiver type) of an unannotated method as if they were annotated as `@PolyTaint` (see Sec. 2.2.3). For calls to such methods, the return value will be treated as `@Tainted` only if at least one of the actual parameters at the call is `@Tainted`. Consider this expression from line 8 from Fig. 1:

```
Paths.get(parent).toAbsolutePath().toFile();
```

The invoked `toFile` method is unannotated, so with our polymorphic treatment, its return value will only be treated as `@Tainted` if the result of the `toAbsolutePath` call is `@Tainted`. Since `toAbsolutePath` is also unannotated, the process recurses, and the taintedness depends on the result of the `Paths.get(parent)` call. Finally, since `Paths.get` is also unannotated, TaintTyper concludes the taintedness of the whole expression depends on whether *`parent`* is tainted, as desired for this example.

An overview of the logic for determining the return type qualifier for a method call, and when to apply polymorphic defaulting, is given in Fig. 3 (the logic for parameter types is similar). The flowchart handles a call to some method `a.b.C#foo`, where `a.b` is the package containing class `C`. If `foo`'s declared return type contains a generic type variable, we use the standard type checking rule for the call site, even if `foo` is in unannotated code. This exception is important since the return type is at least partially determined by type arguments from the call site. E.g., line 6 of Fig. 1 invokes `Map.get`, whose return type is the type variable `V` for map values. Though `Map.get` is unannotated, TaintTyper applies standard checking, to preserve information from the type argument `@Untainted String` given for this particular `Map` at line 1.

TaintTyper also always uses the standard call site type if `foo`'s type been specified in a stub file [3], as such files allow for externally providing types for library routines like sources and sinks. Otherwise, to determine whether code should be treated as annotated, TaintTyper takes as input an *AnnotatedPackages* regular expression (borrowing from the NullAway nullness checker [12]) which specifies which Java packages should be treated as annotated code. Note that this setting does not distinguish between first-party and third-party code, providing flexibility. E.g., when adopting TaintTyper, source packages can be set as annotated gradually, initially leaving other source packages as unannotated and unchecked. Only if the `a.b` package is not part of the annotated packages, the polymorphic handling for unannotated code is applied.

The polymorphic-by-default handling of unannotated code is not sound for all cases. For example, if an unannotated method is itself a tainted source, then its return value should be treated as tainted, independent of the argument taintedness. Similar issues arise for unannotated sink methods. As discussed in Sec. 2.1, we view discovery of source and sink methods as a separate problem from the core checking and inference issues we address here. A method may also return tainted data even with untainted arguments, if a setter method of the receiver is invoked earlier with tainted data. When discovered, such cases can be addressed via stub file models [3]. In practice, our evaluation (Sec. 7) showed that this technique did not lead to false negatives in comparison to two state-of-the-art tools on real-world benchmarks, and that it dramatically increased practicality compared to [2].

4.2 Other Constructs

TaintTyper computes a default `@Untainted` type for a variety of language constructs that are treated as `@Tainted` by the previous checker [2]. Enum constants, class literals, lambda expressions, and fields of annotations are always treated as `@Untainted`. A static final field is treated as `@Untainted` if its initializer expression is `@Untainted`. An array initializer expression (e.g., `new String[] {x,y}`) has `@Untainted` contents by default if all the initial array values are `@Untainted`. Similarly, a cast expression is `@Untainted` if the casted expression is `@Untainted`. In some cases, like static final fields, an explicit `@Untainted` annotation could be written, but our defaulting reduces the annotation burden. Note that other cases like lambda expressions cannot be directly annotated, and can only be handled with a warning suppression without our approach.

`java.util.Collection` data structures are widely used in Java programs, making special-case handling useful. A `Collection` is often constructed directly from another `Collection` or array, e.g., `new ArrayList<>(otherList)`. In such cases, if the other `Collection` holds `@Untainted` elements, TaintTyper always treats the new `Collection` as having `@Untainted` elements. The version of [2] we compared with did not handle these cases correctly due to type inference limitations.[2]

The `Collection.toArray(T[])` method, which converts a `Collection` to a `T[]` array, is also frequently used. With the baseline checker, even for a `Collection` of `@Untainted` values, the array contents type of a `toArray` call is `@Tainted` unless the `toArray` argument is explicitly annotated (e.g., `c.toArray(new @Untainted String[0])`). TaintTyper does not require this annotation and treats the result of `toArray` as having `@Untainted` contents if the `Collection` contents are `@Untainted`.

5 Inference

In this section we detail how TaintTyper performs inference. TaintTyper extends a previous search-based inference technique with support for generic types, `@PolyTaint` annotations, and unannotated code. It also introduces a new optimization that significantly improves inference performance.

[2] The inference limitations have been addressed in more recent versions, but we found that these versions introduced new bugs, so we did not update the version that we compared to.

5.1 Baseline Algorithm

Our inference extends the search-based inference approach of Karimipour et al. [30]. The approach aims to infer annotations that minimize the final number of errors reported by a checker, which in [30] was the nullness checker NullAway [12]. By minimizing the final number of errors, this approach infers annotations that *maximize* the amount of code that fully passes the checker. A recent study showed this search-based approach to work better than alternate approaches in practice [28].

The search performs *black-box inference*, in which the effectiveness of annotations in reducing errors is measured by running the checker itself and observing its output. This technique computes a set of annotations that fix some reported checker errors. Then, the checker is re-applied to see if these annotations cause new errors. Consider this example:

```
1 void m1() { m2(source()); m2(source()); }
2 void m2(String t) { sink(t); }
```

Initially, TaintTyper reports an error at line 2, since `t` is `@Tainted` by default and passed to a sink. A local fix for this error is to annotate `t` as `@Untainted`. But, with this fix, TaintTyper reports two new errors on line 1 (since `source`'s return is `@Tainted`), increasing the total number of errors. So, the fix is rejected by the search. The previous work describes optimizations to speed the search process, by evaluating independent annotations simultaneously [30]; we evaluate the effectiveness of these optimizations for taint inference in Sec. 7.2.3.

The previous work only described how to fix errors reported by NullAway. For tainting, computing annotations for fixes is significantly more complex due to the need to support generic types and `@PolyTaint`. Recent work on pluggable type inference left open inference for generic types due to the challenges involved [30, 31]. In Sec. 5.2, we describe our novel technique for computing annotations to fix tainting errors, supporting generic type arguments and `@PolyTaint`.

5.2 Computing Fixes

For our tainting checker, there are two main causes of reported errors: type incompatibility at a (pseudo-)assignment, and incorrect method overriding (see Sec. 2.2). For both cases, fixing the error requires *adjusting the type* of relevant variables or expressions. E.g., for the example of Sec. 5.1, the initial fix was to change the type of parameter `t` to `@Untainted String`. Alg. 1 gives our new technique to compute the annotations to achieve a desired type adjustment. The main procedure FindAnnots takes as arguments an expression e and a desired type for the fix T_f. It either returns a set of annotations that modify e's type to be T_f, or $\perp$ to indicate it has failed to find such a set. FindAnnots relies on various other procedures, some of which we elide for brevity but describe in text. We also only show handling of key representative language constructs for simplicity. We first explain the basic cases for Alg. 1, and then present our novel handling of generic type arguments and `@PolyTaint` annotations.

5.2.1 Basics

For a variable v (which could be a local, parameter, or field), line 3 uses a routine UpdateType (not shown) to update v's declared type directly. UpdateType may fail and return $\perp$; e.g., TaintTyper does not attempt to convert a raw type like `List` to a generic type like `List<@Untainted String>`, as this change is out of scope for our work (other tools can

Algorithm 1 Pseudocode for finding annotations for a fix.

1: **procedure** $\textsc{FindAnnots}(e, T_f)$
2: **if** e is a variable v **then**
3: **return** $\textsc{UpdateType}(v, T_f)$
4: **else if** e is a binary expression $e_1\, op\, e_2$ **then**
5: **return** $\textsc{FindAnnots}(e_1, T_f) \uplus \textsc{FindAnnots}(e_2, T_f)$
6: **else if** e is a call $e_1.m(e_2, \ldots, e_n)$ **then**
7: $G \leftarrow \textsc{GenericsAnnots}(e, T_f)$
8: **if** $G \neq \bot$ **return** G
9: $P \leftarrow \textsc{PolyTaintAnnots}(e, T_f)$
10: **if** $P \neq \bot$ **return** P
11: $m \leftarrow \textsc{InvokedMethod}(e)$
12: **if** $\textsc{TreatAsPolyTainted}(m)$ **then**
13: **return** $\biguplus_{i \in \textsc{PTArgs}(m)} \textsc{MakeUntainted}(e_i)$
14: **else**
15: **return** $\textsc{UpdateType}(\textsc{ReturnType}(m), T_f)$
16: **end if**
17: **end if**
18: **end procedure**
19: **procedure** $\textsc{MakeUntainted}(e)$
20: $T' \leftarrow$ `@Untainted` $\textsc{TypeOf}(e)$
21: **return** $\textsc{FindAnnots}(e, T')$
22: **end procedure**
23: **procedure** $\textsc{GenericsAnnots}(e, T_f)$
24: $m \leftarrow \textsc{InvokedMethod}(e)$
25: $S \leftarrow \textsc{FindTypeSubst}(\textsc{ReturnType}(m), T_f)$
26: **if** $S = \bot$ **return** $\bot$
27: $e_r \leftarrow \textsc{ReceiverArg}(e)$
28: $T' \leftarrow \textsc{ApplySubst}(S, \textsc{ReceiverType}(m), \textsc{TypeOf}(e_r))$
29: **return** $\textsc{FindAnnots}(e_r, T')$
30: **end procedure**

be applied for such cases [43]). For binary operators with sub-expressions e_1 and e_2, we recursively compute fixes for e_1 and e_2 and then combine them using a special $\uplus$ operator (line 5). $\uplus$ propagates the failure value $\bot$ – it is defined as follows:

$$A \uplus B = \begin{cases} \bot & A = \bot \vee B = \bot, \\ A \cup B & \text{otherwise.} \end{cases}$$

Method calls, handled at lines 6–16, require the most complex handling. We first attempt to handle the call by inferring annotations on generic type arguments (line 7). If that fails, we attempt inference of `@PolyTaint` annotations (line 9). We will explain the generics and `@PolyTaint` cases further shortly. If both generics and `@PolyTaint` inference fail (by returning $\bot$), we fall back to a more direct handling, depending on the method being invoked.

Line 12 checks if the invoked method m should be treated as having `@PolyTaint` annotations. This check returns true if either m has declared `@PolyTaint` annotations or if m is unannotated and our default polymorphic handling applies (see Sec. 4.1). In such cases, to make the call's type untainted, *all* `@PolyTaint` parameters must be made untainted, reflected in line 13 ($\textsc{PTArgs}(m)$ returns the `@PolyTaint` argument positions for m). $\textsc{MakeUntainted}$ (lines 19–22) updates the type of expression e with a top-level `@Untainted` annotation and then recursively calls $\textsc{FindAnnots}$. Any failure to make an argument untainted is propagated using $\uplus$. Finally, for calls to annotated methods, our base handling is to update the declared return type of the invoked method (line 15).

5.2.2 Generics

For calls to methods involving generic types, our approach aims to find fixes that annotate type arguments instead of directly updating the invoked method's return type. To see why, consider the call `paths.get(name)` on line 6 in Fig. 1. This call invokes `Map.get`, whose return type is generic type variable `V`. Say that FindAnnots aims to make the result of the call `@Untainted`. The baseline technique for method calls (line 15 in Alg. 1) would change `get`'s return type to `@Untainted V`, a valid fix. But, this change would force *all* calls to `Map.get` to return untainted values, preventing *any* `Map` from holding possibly-tainted values, which is impractical.

Instead, our technique tries to find a fix that leverages generic type arguments, as shown in the GenericsAnnots routine of Alg. 1. First (line 25), we call FindTypeSubst (not shown) to find a *substitution* for the type variables in the return type of m that yields the desired type T_f. For the above example, GenericsAnnots is called with $e =$ `paths.get(name)` and $T_f =$ `@Untainted String`. Since `get`'s return type is type variable `V`, line 25 successfully finds a substitution $S =$ `V` $\mapsto$ `@Untainted String` that yields T_f. FindTypeSubst may fail to find a substitution, in which case it returns $\perp$ (line 26). FindTypeSubst works by recursing through type structures, mapping type variables to the desired type arguments; we elide details as they are straightforward.

When we successfully find a substitution S, we then *apply* S to the declared receiver type of the method, and recursively try to find corresponding annotations for the receiver argument of the call (lines 27–29). If S does not cover all type variables in the receiver type, we reuse the generic type arguments from the receiver argument at the call site. For our `paths.get` example, the declared type of the `Map.get` receiver is `Map<K,V>`, but our substitution `V` $\mapsto$ `@Untainted String` does not cover `K`. So, we reuse the `String` type argument for `K` from line 1 of Fig. 1, yielding a recursive call FindAnnots(`paths`, `Map<String,@Untainted String>`) at line 29.

The recursive nature of FindAnnots successfully handles much more complex uses of generic types, e.g.:

```
void foo(Map<Integer, +@Untainted String> t) {
  sink(t.values().iterator().next());
}
```

FindAnnots aims to make the return type for the `next` call `@Untainted String`, but it is not immediately evident which generic type argument must be annotated to achieve this. In our algorithm, the generics logic proceeds by recursively trying to make the `iterator` call return `Iterator<@Untainted String>`. This in turn leads to trying to make the `values` call return `Collection<@Untainted String>`, which finally leads to successfully adjusting the type of `t` to `Map<Integer, @Untainted String>`. TaintTyper can also annotate generic type arguments in `extends` clauses of class declarations, and generic methods (where the type variable is scoped to the method instead of the class) are also handled fully.

5.2.3 `@PolyTaint` inference

As noted in Sec. 2.2, `@PolyTaint` can be useful when a method is generic in its tainting behavior but was not declared using generic type variables. Due to the lack of type variables, inference of `@PolyTaint` annotations must *discover* relevant data flow from parameters to return values, which may occur through callee methods. E.g., for the `parentPath` method in Fig. 1, the taintedness of the `path` argument influences the return taintedness via a call to `path.substring`.

Algorithm 2 Pseudocode for inferring polymorphic annotations.

```
procedure PolyTaintAnnots(e, T_f)
    m ← InvokedMethod(e)
    Result ← ∅
    F_return ← FindAnnotForReturnStatements(m, T_f)
    Worklist ← F_return
    Processed ← ∅
    while Worklist ≠ ∅ do
        F_element ← Worklist.pop()
        if F_element is not on a local variable then
            Result ← Result ∪ {F_element}
            continue
        end if
        if F_element ∈ Processed then
            continue
        end if
        Processed ← Processed ∪ {F_element}
        F_assign ← FindAnnotForAssignments(m, F_element, T_f)
        Worklist ← Worklist ∪ F_assign
    end while
    F_Parameters ← {F | F ∈ Result ∧ F is on parameter of m}
    F_NonParameters ← Result \ F_Parameters
    if F_Parameters = ∅ then
        return MakeUntainted(m)
    else
        PolyMethodFix ← MakePolyTainted(m, F_Parameters) ∪ F_NonParameters
        F_args ← ∅
        for arg ∈ PolyMethodFix.args do
            F_args ← F_args ∪ FindAnnots(arg, UpdatedTarget(T_f))
        end for
    end if
    return F_args ⊎ PolyMethodFix
end procedure
```

The PolyTaintAnnots procedure for inferring `@PolyTaint` annotations (called at line 9 in Alg. 1) is conceptually simple: it works by inserting `@PolyTaint` annotations, observing where such annotations lead to type checking errors, and then recursively inserting more annotations to fix those errors. However, we found that a straightforward implementation based on this strategy was too inefficient, so we introduced two improvements. First, a naïve approach to discovering new type errors is to re-run the full type checker, but leads to many expensive checker runs; instead, we implemented our own limited analysis of data flows relevant to `@PolyTaint` to discover new errors. Second, we bounded the depth of the search into callee methods, giving up and returning $\perp$ if inference required searching further (we found depth five to work best in our experiments).

Alg. 2 gives a simplified view of the PolyTaintAnnots procedure. The algorithm starts by invoking FindAnnotForReturnStatements (not shown), which scans the method body for returned expressions and uses FindAnnots to discover the annotations needed to align each expression's type with T_f. If any of the computed annotations targets a local variable v, PolyTaintAnnots uses FindAnnotForAssignments (not shown) to scan for expressions assigned to v and compute necessary annotations for those expressions (again via FindAnnots). This process iterates using a worklist until all locals are handled. Upon completing the iterations, if there exists an annotation on a method parameter, the method is identified as polymorphic, and the necessary annotations are computed. The method indirectly invokes FindAnnots from Alg. 1 through calls to FindAnnotForReturnStatements and FindAnnotForAssignments, which may recursively invoke PolyTaintAnnots. A separate depth bound ensures termination, and the algorithm returns $\perp$ if the bound is reached or if any call to FindAnnots returns $\perp$. When a method is determined to be polymorphic, the algorithm calculates the required annotations for its arguments and combines these with the annotations derived for the return type.

5.2.4 Example

We now discuss the overall process of applying inference to our motivating example in Fig. 1. Since the error in the unannotated code is reported at line 8, initially FindAnnots is invoked to try to make the `toFile()` call passed to `sink` have type `@Untainted String`. As discussed in Sec. 4.1, this expression includes nested calls to unannotated code, handled by line 13 in Alg. 1. Eventually, this leads to annotating `parent` as `@Untainted`. This annotation causes a new checker error at line 7, leading inference to use FindAnnots to make the `parentPath(path)` call `@Untainted`. Here, our `@PolyTaint` inference succeeds for `parentPath`, leading to the annotations on line 2. The search then makes `path @Untainted`, causing a type error at line 6. Here, FindAnnots makes the `paths.get` call `@Untainted` by updating the generic type of the `paths` field, discussed in detail in Sec. 5.2.2 above. With this change, no new errors are reported, completing inference.

5.2.5 Local Variable Optimization

Our initial inference implementation took an excessive amount of time, nearly 24 hours for larger benchmarks. Most inference time is spent running the checker to detect the impact of annotations on warnings. For tainting, we found that many such checker runs were for annotations on local variables, e.g., the runs for `parent` and `path` in Fig. 1 (discussed above). As an optimization, we enhanced our fix computation to *internally* determine the impact of local variable annotations rather than using the checker. This reasoning requires finding assignments to the relevant locals, and then recursively invoking FindAnnots to make the type of each assignment's right-hand side match the local's new type, similar to the logic shown in Alg. 2 for inferring `@PolyTaint`. With this optimization, two checker calls (for `parent` and `path`) are eliminated in inference for Fig. 1. In Sec. 7.2 we show this optimization has a very significant impact on inference performance.

6 Implementation

TaintTyper includes both type-based taint checking and inference (see Fig. 2). TaintTyper's checker (see Sec. 4) was built using the Checker Framework [36, 19], version 3.42.0. Use of the Checker Framework equips the TaintTyper checker with robust support for flow-sensitive local type inference, checking of generic types, and qualifier polymorphism. For our prototype, we modeled a number of source and sink methods involved in the most common Java vulnerabilities [7]. Our sinks include common methods that write to a file, send data over the network, or execute sensitive system or database commands. We modeled sources that read from the network, the file system, or user input. We also modeled a few relevant well-known sanitizer methods [8]. As noted in Sec. 2, creating more complete databases of sources, sinks, and sanitizers is still a research problem, and TaintTyper can easily benefit from further advances in that area.

TaintTyper's inference implementation uses a modified version of the NullAwayAnnotator tool [30]. Inside the TaintTyper checker, we implemented Alg. 1 to find annotations that can fix an error, and added support for serializing this information in the checker output. Our inference implementation reads in this serialized output to use during its search. The search is similar to that of [30], enhanced with our local variable optimization (Sec. 5.2.5). The search is depth bounded [30], and we used a bound of 15 in our experiments.

7 Evaluation

Here we present an experimental evaluation showing the high effectiveness of TaintTyper in practice.

7.1 Experimental setup and research questions

From previous work we found three benchmark suites will all tainting violations labeled, serving as a ground truth. They are:

- **Securibench Micro** [32], which provides 122 servlets exhibiting potential information-flow vulnerabilities, with the source code annotated with benign or problematic flows.
- **JInfoFlow** [24], a self-contained benchmark of 12 information-flow violations featuring reflection-intensive, event-driven code without dependencies on external libraries.
- **Injection Experiments** [39], comprising 8 Java programs with information-flow violations reported by their tool. While the original tool is no longer available, the dataset remains accessible. In the metadata, the authors label the reports from their tool as true or false positives.

Although these three benchmarks capture various interesting cases, they consist either of toy examples [32, 24] or projects that have not been maintained for years [39]. Consequently, we use them solely as one validation of the impact of the techniques of Sec. 4 on TaintTyper's recall.

For a more realistic evaluation, we examine a suite of actively-developed open-source Java programs used in recent work [10, 14]. We selected only projects for which there were some reported tainting errors. We also strove to include a variety of project sizes and types, ensuring the benchmark reflects real-world scenarios; our suite includes a web framework, content management system, security framework, forum software, and library management system. Due to the lack of a ground truth on these benchmarks, we made considerable manual efforts to ensure the accuracy of our results. This included carefully comparing results against previous static analyzers and manually annotating the code for comparison (further details below). In the end, we prioritized benchmarks that were the most representative and important, ultimately selecting seven projects for our evaluation, as listed in Table 1. We used all benchmarks from [10] for which there were reported tainting errors except for webgoat and opencms, which were not included due to complex build system configurations that TaintTyper cannot yet handle. One such complexity arises when the code relies on generated code, such as when using Project Lombok [1]. If a generated getter method is inferred to return untainted, it must first be delomboked and explicitly annotated. However, when the code is compiled, the generated code is overwritten, causing the loss of information needed to propagate the untainted annotation from the getter to the corresponding field. This is not a fundamental limitation of our approach but rather an implementation challenge that could be addressed with additional engineering effort.

Table 1 Benchmark sizes and inferred annotation counts.

Project	KLoC	Inferred Annotations	
		Total	per kLoC
esapi-java-legacy	18.3	265	14.4
pybbs	9.7	105	10.8
alfresco-core	13.4	81	6.0
alfresco-remote-api	85.8	213	2.5
cxf	47.5	296	6.2
struts	49.3	313	6.3
commons-configuration	20.2	261	12.9

We used CodeQL [4] v2.15.1 and P/Taint [24, 5] as baseline tools for our evaluation. CodeQL is a production-quality security analysis tool, widely used and freely available. P/Taint uses state-of-the-art whole-program analysis techniques, also employed in recent work [13]. We configured both tools to detect issues involving the sources and sinks modeled for TaintTyper (Sec. 6). We considered a variety of other tools for our experiments but found them unsuitable. FlowDroid [11] is a well-known taint analyzer for Android, but it does not support analysis of Java server programs. And, the taint analyses in SonarQube [9], RAPID [21], and CompTaint [13] are not freely available.

Given these benchmarks and tools, we studied seven research questions:

RQ1 Does TaintTyper find the known errors in existing labeled benchmarks?

RQ2 After inference, how do TaintTyper's reported errors compare to those reported by state-of-the-art tools (in terms of precision and recall)?

RQ3 After inference, how does TaintTyper's checking time compare with state-of-the-art tools on real-world benchmarks?

RQ4 Does TaintTyper's inference run in a reasonable amount of time, and are our optimizations effective?

RQ5 Does TaintTyper require a reasonable number of annotations?

RQ6 How do the annotations inferred by TaintTyper compare to manually-written annotations?

RQ7 How is TaintTyper's effectiveness impacted if we disable checker improvements, generic type argument inference, and `@PolyTaint` inference?

We address RQ1–RQ4 in Sec. 7.2. Then, Sec. 7.3 addresses RQ5 and RQ6, and Sec. 7.4 answers RQ7. All experiments were conducted in a Google Cloud instance with an AMD EPYC Milan 3rd Generation 2.45GHz CPU with 32vcpu (16 cores) and 128GB memory.

7.2 Inference effectiveness

7.2.1 Soundness on labeled benchmarks

To evaluate the soundness of TaintTyper, the three labeled benchmark suites were used. For these experiments, we customized the source and sink specifications used by TaintTyper to match what was expected by each benchmark suite.

We first ran TaintTyper's checker without inference, and confirmed that it did not miss *any* labeled vulnerabilities in the benchmarks. We then applied our inference to annotate the benchmarks automatically and re-check them again, to check how the inferred annotations impacted recall.

Table 2 Without inference, TaintTyper reports all the labeled true-positive issues across these three benchmarks. With inferred annotatations, it misses one issue in Securibench Micro, detects all issues in JInfoFlow, and misses 15 issues in Injection Experiments.

SecuriBench Micro		JInfoFlow-bench		Injection Experiments	
Total	136(137)	Total	12	Total	730(745)
aliasing	12	JInfoFlow/basic	2	Snake&Ladder	40
arrays	9	JInfoFlow/ctx	5	MediaPlayer	0
basic	61	JInfoFlow/event	5	EmergencySNRest	0(3)
collections	14			FarmTycoon	5
datastructures	5			Abagail	0
factories	3			JExcelAPI	3(4)
inter	16			Colossus	682(693)
pred	5				
reflection	4				
sanitizers	3(4)				
session	3				
strong updates	1				

Table 3 Precision and recall results across our benchmark suite.

Project	TP	CodeQL		TaintTyper	
		Precision	Recall	Precision	Recall
esapi-java-legacy	18	0.90	0.50	0.95	1.00
pybbs	9	1.00	0.89	1.00	1.00
alfresco-core	2	1.00	0.50	1.00	1.00
alfresco-remote-api	21	0.82	0.43	0.70	1.00
cxf	9	1.00	0.11	0.75	1.00
struts	23	0.39	0.39	0.50	1.00
commons-configuration	11	1.00	0.73	0.69	1.00

For Securibench Micro, the metadata indicated 136 labeled vulnerabilities. However, manual inspection revealed 137 in total, with one unlabeled issue in *Basic26.java* and one missing label in *Basic31.java*. Securibench Micro also makes extensive use of raw types, which appear rarely in modern Java code and are not yet fully handled by TaintTyper; we inserted (unannotated) generic type arguments for these cases. With these fixes, after inference, TaintTyper only missed one true issue out of 137; this was due to an interaction between our polymorphic-by-default library handling and side effects, as discussed in Sec. 4.1.

In inference-annotated JInfoFlow-bench, TaintTyper successfully identifies all 12 labeled vulnerabilities. The Injection Experiments consist of eight Java programs, one of which could not be compiled. We annotated the remaining seven programs (collectively comprising 754 labeled issues) using TaintTyper inference. Eight of the labeled issues occur in files not part of the available benchmark codebase, and one occurs in test code. Upon manually inspecting the inferred annotations and the reported errors from TaintTyper, we found that it discovered 730 of the remaining 745 issues, with the missed issues again due to side-effecting third-party library calls. The number of missed issues across these benchmarks is small, and as shall be shown in Sec. 7.2.2, we saw *no* missed issues for TaintTyper when compared to a production-quality tool on real-world benchmarks. Table 2 summarizes our findings across these benchmarks.

RQ1: TaintTyper's checker identifies all known issues in the labeled benchmarks, and inference reduces recall only slightly.

Table 4 Analysis runtime and inference runtime results across our benchmark suite.

Project	Analysis Runtime			Inference Runtime			
	TaintTyper	CodeQL	P/Taint	ALL	LOC	CORE	NONE
esapi-java-legacy	19s	94s	53m	23m	65m	47m	105m
pybbs	12s	83s	35m	8m	15m	11m	22m
alfresco-core	15s	75s	21m	9m	14m	20m	31m
alfresco-remote-api	69s	202s	167m	134m	311m	391m	711m
cxf	47s	1076s	151m	148m	294m	498m	817m
struts	39s	425s	>48h	343m	629m	1354m	>48h
commons-configuration	25s	114s	209m	105m	168m	299m	432m

7.2.2 Reported errors

Table 3 compares the precision and recall of TaintTyper and CodeQL for our benchmarks, addressing RQ2. For TaintTyper, the errors are computed after inference has run, so the code includes inferred annotations. Computing recall requires knowing the ground truth of all real issues in these benchmarks, which is infeasible to collect. To estimate recall, we use the union of all true positive issues reported by TaintTyper, CodeQL, and P/Taint as our ground truth. (We treat a report as a true positive if the corresponding dataflow is deemed feasible by manual inspection.) This may over-estimate the true recall of all tools, but it provides a good basis for comparing the tools. The “TP” column gives the number of true positive issues for each benchmark.

Comparing error reports between the tools was non-trivial due to their different reporting approaches. For TaintTyper, an error is typically reported as a code location where a `@Tainted` value is being written into an `@Untainted` location. CodeQL reports an error as a data flow trace from a source to a sink, and P/Taint reports an error as a source/sink pair without a trace. Comparing these errors required manually matching each true-positive TaintTyper error to a step in some CodeQL trace, or to some data flow for a P/Taint source/sink pair, and vice versa. CodeQL sometimes treats a formal parameter as a tainted source, without any explicit call passing in tainted data. In these cases, TaintTyper annotates the parameter as `@Untainted`, capturing the fact that tainted data should not be passed in, but does not report an error. We count such cases as equivalent to reporting the error; TaintTyper could easily report errors for such cases if desired.

We exclude the precision and recall of P/Taint from Table 3 as both were very low on our benchmarks. P/Taint found a total of six true positive issues across the benchmarks (all of which were also found by some other tool), and it reported 50 false positives. We carefully checked our P/Taint configuration and confirmed it found expected issues in smaller benchmarks like Securibench Micro [32]. We also consulted with the tool authors, who acknowledged that P/Taint may not handle these benchmarks well (e.g., due to missing framework support). Still, triaging P/Taint results was very useful, to gain further confidence in our ground truth.

TaintTyper finds all true positive issues discovered by CodeQL and P/Taint, leading to a recall of 1 on all benchmarks in Table 3. TaintTyper also finds additional true positives missed by CodeQL, reflected in CodeQL’s lower recall numbers. TaintTyper has lower precision than CodeQL on three benchmarks; we suspect this is due to heuristics in CodeQL that are missing in TaintTyper. Still, TaintTyper matches or exceeds CodeQL’s precision on the other four benchmarks.

Table 5 Number of annotations inserted manually and by TAINTTYPER inference, and final error counts with various features disabled; C for checker improvements, G for generics inference, and P for `@PolyTaint` inference.

Project	Manual Annotation Count			Inferred Annotation Count			Error Count			
	Total	TypeArg	PolyTaint	Total	TypeArg	PolyTaint	C off	G off	P off	All Active
esapi-java-legacy	278	20	34	265	13	34	49	16	13	13
pybbs	95	27	4	105	46	4	21	16	2	2
alfresco-core	81	38	0	81	36	0	12	5	2	2
alfresco-remote-api	110	12	10	213	22	7	66	42	27	24
cxf	380	62	37	296	51	42	73	37	20	11
struts	347	50	5	313	47	7	116	54	44	37
commons-configuration	239	10	25	261	12	23	33	17	11	11

RQ2: On our benchmarks, TAINTTYPER has outstanding recall, with comparable precision to CodeQL.

7.2.3 Performance

Table 4 gives analysis runtimes for TAINTTYPER, CodeQL, and P/Taint checking. The speedups of TAINTTYPER's checker over CodeQL are quite significant, ranging from 2.93X–22.9X. And, we see orders-of-magnitude speedups compared to P/Taint, which could not analyze the struts benchmark within a 48-hour time limit. Given that TAINTTYPER's checking is modular and incremental, we expect even larger speedups over the whole-program analysis approach for larger benchmarks. As a sanity check of the benefits of incremental checking, we manually re-ran TAINTTYPER checking for five randomly-chosen source files for alfresco-remote-api and struts (the two largest benchmarks), and observed further speedups of 8.7X and 10.9X respectively.

RQ3: TAINTTYPER's checker runs much faster than the baseline tools, with further incremental speedups.

Table 4 shows inference performance with all optimizations enabled in the ALL column. Fully-optimized inference always ran in less than 6 hours, suitable for an overnight run and acceptable since it only needs to run once (see Fig. 2). The LOC, CORE, and NONE columns respectively show running times with our new local variable optimization only (Sec. 5.2.5), the core optimizations of [30] only (see Sec. 5.1), and no optimizations. The core optimizations of [30] seem to have a lesser impact for tainting inference than they did for nullability; excluding struts (which did not terminate in 48 hours with optimizations disabled), we see a maximum speedup of 2.23X, compared to 17.8X reported in [30]. Our local variables optimization on its own yields speedups of 1.47X–2.78X, sometimes larger than the core optimizations. Fortunately, the techniques are complementary, together yielding the best speedups of 2.75X–5.52X.

RQ4: TAINTTYPER has acceptable inference performance, aided significantly by our new optimization.

7.3 Assessing annotations

Table 1 shows the number of annotations inferred by TaintTyper for our benchmarks and the corresponding annotation density, addressing RQ4. The number of inferred annotations per KLoC is relatively low, ranging from 2.5-14.4. In comparison, Banerjee et al. [12] reported an average of 13 annotations per KLoC for NullAway, ranging up to 46 annotations, and that checker has been widely adopted.

RQ5: The annotation requirements of TaintTyper are low enough to enable adoption, given the importance of preventing tainting vulnerabilities.

Table 5 compares the number of annotations inferred by TaintTyper to the number of annotations inserted in a separate manual process. Two co-authors added manual annotations, each checking the other's work and coming to consensus for any disagreement. We completed this process, adding 1,530 manual annotations. We limited the manual changes to inserting annotations, prohibiting code changes. This limitation is contrived: a developer would most likely fix bugs and refactor code alongside adding annotations. We chose this methodology to fairly compare with TaintTyper inference, which only inserts annotations.

Table 5 shows that the number of inserted annotations does tend to differ between the two approaches. We inspected the discrepancies in detail, and found that in all cases, TaintTyper's inserted annotations were reasonable; which annotations were "better" was a subjective question. In cases where TaintTyper inserted fewer annotations, one pattern was where a manually-written annotation captured some desired invariant, but TaintTyper eschewed the annotation since it increased the final error count. A second pattern were cases where during manual annotation, an opportunity to use `@PolyTaint` was missed, but TaintTyper made use of `@PolyTaint` to avoid several other `@Untainted` annotations.

For cases where TaintTyper inserted more annotations, a common explanation was again reducing error count; TaintTyper would insert many extra annotations to reduce the final error count by one, but this was not deemed to be worthwhile during manual annotation. In such cases, arguably a better fix would be to restructure the code so fewer annotations would be needed. We could easily add a setting to TaintTyper to limit the number of annotations it inserts to fix a single warning to avoid such cases. In an extended version of the paper [29], we give detailed examples illustrating the above scenarios.

RQ6: TaintTyper's inserted annotations were always acceptable for insertion into source code, and sometimes improved on our manual annotations.

7.4 Ablation

Finally, to answer RQ6, we performed an ablation study to see how many errors are reported by TaintTyper when disabling the checker features of Sec. 4, generics inference (Sec. 5.2.2), and `@PolyTaint` inference (Sec. 5.2.3) individually. The results are shown in Table 5. We see large increases in reported errors with our new checker features disabled (3X-10.5X more errors), and similar impacts with generics support disabled (1.2X-8X more errors), showing the criticality of these features for precision. Further, without the new checker features, 85 of the additional errors were false positives that could not be removed with annotations (see Sec. 4.2); such cases require a warning suppression, causing developer frustration.

Inference of @PolyTaint has a less significant impact on final error counts. However, @PolyTaint inference is still critical for generating annotations close to what would be manually written, as shown in Table 5. Manual annotations included a total of 115 @PolyTaint annotations across our benchmarks, and with inference of such annotations disabled, none of these would be found by TaintTyper. Similarly, Table 5 shows that generic type annotations are used commonly, and hence TaintTyper's support for inferring these annotations is important.

RQ7: Our new checker features, generic type support, and @PolyTaint support were all critical to TaintTyper's effectiveness.

7.5 Threats to Validity

Our benchmark choices are a threat to external validity. We described our methodology for choosing benchmarks in Sec. 7.1. While we strove to choose diverse benchmarks, it is possible that TaintTyper will be less effective on a different set of programs. Our choice of tools for comparison (CodeQL and P/Taint) is another threat to external validity; other whole-program static analyzers may perform differently. See Sec. 7.1 for our process in choosing these tools. Paper co-authors performing the manual annotation of benchmarks is a threat to internal validity. We strove to add these annotations disregarding the workings of TaintTyper's inference. Other developers may manually annotate code in different ways, but a user study evaluating the degree of such differences is out of scope for this work. Implementation bugs in TaintTyper may also impact internal validity; but, we have done significant checking of correctness using manual inspection and a suite of regression tests.

8 Related Work

There is a broad literature on taint analysis; here we discuss the most closely related work. In Sec. 2 we contrasted type-based techniques with whole-program approaches [13, 44, 11, 27, 24, 4, 21, 45, 34, 46, 26], and we compared with CodeQL [4] and P/Taint [24] in our evaluation. Recent work by Banerjee et al. [13] describes an approach to incremental taint analysis but does not evaluate its performance. Szabo [42] presents an initial exploration of incrementalizing CodeQL analyses. While their incremental running times were promising, the additional memory usage of their technique was prohibitive. Type-based taint analysis is naturally incremental and does not suffer from these engineering challenges.

The most closely related whole-program approach is that of Huang et al. [27], which partially inspired our work. Their work is also formulated in terms of type-based taint analysis and type inference. Their type system does not support generic types: instead, they apply polymorphic types to fields to achieve a form of field sensitivity. It is unclear how to expose such field polymorphism in terms of standard pluggable types. Their work does not present a technique to persist types into source code to enable checking without inference and is specific to Android applications so we did not include this tool in our evaluations. TaintTyper performs inference for standard Java pluggable types, including generic types, enabling a straightforward integration with standard development workflows. The inference of [27] runs faster than ours since it operates over a single constraint encoding. However, our inference works with an existing checker without requiring a constraint encoding, which simplifies making improvements to the checker like those of Sec. 4.

There are many other approaches to enforcing information flow properties with types. The well-known Jif system [35] and many subsequent works support sophisticated properties like multiple principals. Such features are not necessary to capture the vulnerabilities targeted by typical taint analyses and this work. Ernst et al. [22] target verification of Android apps, also using pluggable types. Their system aims for soundness for a security-critical military context, leading to an annotation burden of 60 annotations per KLoC, much higher than ours. TaintTyper eschews strict soundness to reduce the annotation burden for usability. We believe TaintTyper's inference technique could be adapted to systems like [22] in the future.

Regarding alternate inference approaches, standard type inference [37, Chapter 22] tries to discover a complete typing for a program in a given type system. Such techniques do not directly apply to our scenario, as a typical program will not be verifiable in our taint type system solely through adding annotations, due to true or false positives; we aim to find a good set of annotations for such untypable programs. Kellogg et al. [31] present a general inference technique for any pluggable type system built on the Checker Framework. However, their technique does not infer polymorphic or generic type annotations, and it may generate many more annotations than what would be written by developer.

Checker Framework Inference [16] uses constraint-based approach to infer types, with applications to domains like a type system for measurement units [47]. We chose a "black box" inference approach [30] for this work since it enabled re-using a checker implementation without reimplementing its logic in a constraint language. The approach of [47] does not support inference of polymorphic method annotations like `@PolyTaint`, and the paper does not discuss in detail its level of support for inferring annotations on generic type arguments.

9 Conclusions

We have presented TaintTyper, a novel approach to type-based taint checking and inference for Java. TaintTyper includes a novel checker that makes the core tainting type system more practical to use, and a novel inference algorithm capable of handling generic types and polymorphic annotations. Our evaluation showed that TaintTyper provided significant scalability advantages over a standard approach, with improved recall and comparable precision, and inferred annotations suitable for direct inclusion in source code. Hence, TaintTyper is a significant step toward practical and widescale type-based taint checking for Java.

References

1 Project Lombok. https://projectlombok.org, 2019. Accessed: 2022-04-03.
2 Checker Framework Manual: Tainting Checker. https://checkerframework.org/manual/#tainting-checker, 2024. Accessed: 2024-03-13.
3 Checker Framework Manual: Using stub classes. https://checkerframework.org/manual/#stub, 2024. Accessed: 2024-03-13.
4 CodeQL. https://codeql.github.com, 2024. Accessed: 2024-02-07.
5 Doop – Framework for Java Pointer and Taint Analysis (using P/Taint). https://github.com/plast-lab/doop, 2024. Accessed: 2024-07-29.
6 Gradle User Guide – Incremental Java compilation. https://docs.gradle.org/current/userguide/java_plugin.html#sec:incremental_compile, 2024. Accessed: 2024-07-29.
7 OWASP Top Ten. https://owasp.org/www-project-top-ten/, 2024. Accessed: 2024-04-11.
8 Sanitizer Methods in OWASP Find Security Bugs. https://github.com/find-sec-bugs/find-sec-bugs/tree/master/findsecbugs-plugin/src/main/resources/safe-encoders/, 2024. Accessed: 2024-04-11.
9 SonarQube. https://www.sonarsource.com/products/sonarqube/, 2024. Accessed: 2024-07-26.

10 Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. Static analysis of java enterprise applications: frameworks and caches, the elephants in the room. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 794–807. ACM, 2020. doi:10.1145/3385412.3386026.

11 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM, 2014. doi:10.1145/2594291.2594299.

12 Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. NullAway: Practical type-based null safety for Java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, 2019. doi:10.1145/3338906.3338919.

13 Subarno Banerjee, Siwei Cui, Michael Emmi, Antonio Filieri, Liana Hadarean, Peixuan Li, Linghui Luo, Goran Piskachev, Nicolás Rosner, Aritra Sengupta, Omer Tripp, and Jingbo Wang. Compositional taint analysis for enforcing security policies at scale. In *ESEC/FSE 2023*, 2023. doi:10.1145/3611643.3613889.

14 Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, pages 464–468. ACM, 2022. doi:10.1145/3524842.3528482.

15 Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pages 3–11, 2015. doi:10.1007/978-3-319-17524-9_1.

16 Checker Framework Inference. https://github.com/opprop/checker-framework-inference, 2022. Accessed: 2022-04-02.

17 Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. Scalable taint specification inference with big code. In *PLDI*, pages 760–774, June 2019. doi:10.1145/3314221.3314648.

18 Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, July 1977. doi:10.1145/359636.359712.

19 Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. Building and using pluggable type-checkers. In *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 2011. doi:10.1145/1985793.1985889.

20 Saikat Dutta, Diego Garbervetsky, Shuvendu K. Lahiri, and Max Schäfer. InspectJS: leveraging code similarity and user-feedback for effective taint specification inference for javascript. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, pages 165–174, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510457.3513048.

21 Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. RAPID: checking API usage for the cloud in the cloud. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 1416–1426. ACM, 2021. doi:10.1145/3468264.3473934.

22 Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward XueJun Wu. Collaborative verification of information flow for a high-assurance app store. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1092–1104. ACM, 2014. doi:10.1145/2660267.2660343.

23 Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, pages 192–203, 1999. doi:10.1145/301618.301665.

24 Neville Grech and Yannis Smaragdakis. P/Taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):102:1–102:28, 2017. doi:10.1145/3133926.

25 Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. An efficient tunable selective points-to analysis for large codebases. In Karim Ali and Cristina Cifuentes, editors, *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, pages 13–18. ACM, 2017. doi:10.1145/3088515.3088519.

26 Wei Huang, Yao Dong, and Ana L. Milanova. Type-based taint analysis for java web applications. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8411 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2014. doi:10.1007/978-3-642-54804-8_10.

27 Wei Huang, Yao Dong, Ana L. Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 106–117. ACM, 2015. doi:10.1145/2771783.2771803.

28 Nima Karimipour, Erfan Arvan, Martin Kellogg, and Manu Sridharan. A new approach to evaluating nullability inference tools. *Proc. ACM Softw. Eng*, 2(FSE), July 2025. doi:10.1145/3715732.

29 Nima Karimipour, Kanak Das, Manu Sridharan, and Behnaz Hassanshahi. Practical Type-Based Taint Checking and Inference (Extended Version). *arXiv*, 2025. doi:10.48550/arXiv.2504.18529.

30 Nima Karimipour, Justin Pham, Lazaro Clapp, and Manu Sridharan. Practical inference of nullability types. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1395–1406. ACM, 2023. doi:10.1145/3611643.3616326.

31 Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, and Michael D. Ernst. Pluggable type inference for free. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 1542–1554. IEEE, 2023. doi:10.1109/ASE56229.2023.00186.

32 Benjamin Livshits. *Improving software security with precise static and runtime analysis.* PhD thesis, Stanford University, Stanford, CA, USA, 2006. AAI3242585.

33 Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 75–86, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1542476.1542485.

34 V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In Patrick D. McDaniel, editor, *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. USENIX Association, 2005. URL: https://www.usenix.org/conference/14th-usenix-security-symposium/finding-security-vulnerabilities-java-applications-static.

35 Andrew C. Myers. Jflow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/292540.292561.

36 Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 2008. doi:10.1145/1390630.1390656.

37 Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

38 Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS Symposium 2014*, January 2014. doi:10.14722/ndss.2014.23039.

39 Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. Static identification of injection attacks in java. *ACM Trans. Program. Lang. Syst.*, 41(3):18:1–18:58, 2019. doi:10.1145/3332371.

40 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In David Clarke, Tobias Wrigstad, and James Noble, editors, *Aliasing in Object-Oriented Programming.* Springer, 2013. doi:10.1007/978-3-642-36946-9_8.

41 Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. Interactive Abstract Interpretation with Demanded Summarization. *TOPLAS*, 46(1), 2024. doi:10.1145/3648441.

42 Tamás Szabó. Incrementalizing production CodeQL analyses. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1716–1726. ACM, 2023. doi:10.1145/3611643.3613860.

43 Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.*, 33(3):9:1–9:47, 2011. doi:10.1145/1961204.1961205.

44 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 87–97. ACM, 2009. doi:10.1145/1542476.1542486.

45 Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. Scaling static taint analysis to industrial SOA applications: a case study at Alibaba. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1477–1486. ACM, 2020. doi:10.1145/3368089.3417059.

46 Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 171–180. ACM, 2008. doi:10.1145/1368088.1368112.

47 Tongtong Xiang, Jeff Y. Luo, and Werner Dietl. Precise inference of expressive units of measurement types. *Proc. ACM Program. Lang.*, 4(OOPSLA):142:1–142:28, 2020. doi:10.1145/3428210.