P/Taint: Unified Points-to and Taint Analysis

NEVILLE GRECH, University of Athens and University of Malta
YANNIS SMARAGDAKIS, University of Athens

Static information-flow analysis (especially taint-analysis) is a key technique in software security, computing where sensitive or untrusted data can propagate in a program. Points-to analysis is a fundamental static program analysis, computing what abstract objects a program expression may point to. In this work, we propose a deep unification of information-flow and points-to analysis. We observe that information-flow analysis is not a mere high-level client of points-to information, but it is indeed identical to points-to analysis on artificial abstract objects that represent different information sources. The very same algorithm can compute, simultaneously, two interlinked but separate results (points-to and information-flow values) with changes only to its initial conditions.

The benefits of such a unification are manifold. We can use existing points-to analysis implementations, with virtually no modification (only minor additions of extra logic for sanitization) to compute information flow concepts, such as value tainting. The algorithmic enhancements of points-to analysis (e.g., different flavors of context sensitivity) can be applied transparently to information-flow analysis. Heavy engineering work on points-to analysis (e.g., handling of the reflection API for Java) applies to information-flow analysis without extra effort. We demonstrate the benefits in a realistic implementation that leverages the Doop points-to analysis framework (including its context-sensitivity and reflection analysis features) to provide an information-flow analysis with excellent precision (over 91%) and recall (over 99%) for standard Java information-flow benchmarks.

The analysis comfortably scales to large, real-world Android applications, analyzing the Facebook Messenger app with more than $55K$ classes in under 7 hours.

CCS Concepts: • **Software and its engineering → General programming languages**;

Additional Key Words and Phrases: Taint Analysis, Pointer Analysis, Android

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

1 INTRODUCTION

Large software systems are ubiquitous and their security is a major concern. Malicious or ill-designed programs can have their operation affected by untrusted inputs or can leak sensitive information, such as private messages, location information, financial details, etc. Static *information-flow analysis* (or "taint analysis")[1] [Arzt et al. 2014; Huang et al. 2015; Lerch et al. 2014; Tripp et al. 2009] addresses this problem and exposes potential violations to security analysts or automated tools by discovering where sensitive information can flow throughout the program.

[1] The term "information-flow analysis" can also be used to describe other analyses, such as dependence analysis [Johnson et al. 2015]. Taint analysis is the most common information-flow analysis, however, and the one most implemented in practice. In the context of this paper, we treat the two terms as synonyms.

Authors' email: me@nevillegrech.com and yannis@smaragd.org.

2475-1421/2017/10-ART102
https://doi.org/10.1145/3133926

Points-to analysis (or "pointer analysis") [Hind 2001; Ryder 2003; Sridharan et al. 2013] is a core static analysis that answers the question "what objects can a program expression refer to"? The analysis can be used as a low-level substrate for a variety of clients: bug detectors [Nikolić and Spoto 2012], code completion engines [Raychev et al. 2014], refactoring transformations [Schäfer et al. 2011], program slicers [Wu et al. 2012], de-obfuscators [Raychev et al. 2015], optimizers [Lattner et al. 2007], and more.

Both information-flow and points-to analysis are *whole-program static analysis techniques*. Developing static analyses of this kind is a highly demanding task of careful and precise modeling of language semantics and complex features. Even the seemingly simple effort of computing a program's call-graph (i.e., which program function can call which other) requires sophisticated analysis in order to achieve precision.

The tenet of our work is that points-to and information-flow can be unified into a single analysis. This is simultaneously an intuitive and an unexpected result. Both analyses compute "where can values flow in a program?" and are thus similar at a high-level. At the same time, the analyses have significant differences: points-to computes where abstract (heap) objects can be referenced; information-flow analysis computes which taint sources can influence a value and whether a value can reach a sink. Notably, taint can be transferred to values of different types (e.g., a tainted byte array can be used to compute a tainted string). Certainly, traditional information-flow analysis implementations are stand-alone and, if the analysis consumes points-to information, it does so as a mere client.

Our techniques are different in this regard. We unify both information and pointer analysis and amalgamate their algorithmic detail. The result is that the same algorithm can then compute both points-to and information flow—at the same time. The key technique is that of *changing the domain of points-to information targets*. Whereas in a standard points-to analysis a variable can point to an abstract object (a.k.a. a *heap allocation*), under our analysis a variable points to more general *values*. Unlike heap allocations, the possible values are not pre-determined in the program text—new ones can be introduced on-the-fly. This introduction of values happens for each taint source, as well as for each operation that propagates a taint. The algorithmic logic remains the same as in traditional points-to analysis: the only difference is that the logic now operates over both real and artificial "objects", with the latter representing taint information.

This approach can be applied to a wide range of points-to analyses, both in practice and in terms of high-level reasoning. We demonstrate the latter by adding information-flow analysis on top of a formal model for a wide range of points-to analyses. Accordingly, we employ the approach in the development of P/Taint: a practical information-flow framework built as an extension of the Doop points-to analysis framework [Bravenboer and Smaragdakis 2009b] for Java. P/Taint adds significant functionality to Doop: for computing tainted values on-the-fly, for marking taint sources, for propagating taint between values of unrelated types, and for value sanitization. However, this functionality is not a client of the Doop points-to analysis: the taint analysis computation is performed by the existing, virtually unmodified Doop algorithms, fed with augmented input. P/Taint showcases several benefits of the approach:

- Although the Doop implementation consists of several thousands of logical rules, in some tens-of-thousands of lines of code, the development of P/Taint has only required changes to a handful of rules of existing Doop code. The rest of the P/Taint functionality is a clean add-on, applicable orthogonally to the many analyses of the framework, without replicating any of the points-to analysis machinery for dealing with different language features.
- P/Taint inherits all of the rich analyses of Doop and can use them to gain *precision* in information-flow analysis, i.e., to reduce *false positives*. Specifically, P/Taint can transparently use any of several tens of analyses, with different kinds of context sensitivity (e.g., call-site

sensitivity [Sharir and Pnueli 1981; Shivers 1991], object-sensitivity [Milanova et al. 2002, 2005], type-sensitivity [Smaragdakis et al. 2011]). The use of context is valuable for precise taint propagation, in exactly the same way as it is for points-to analysis.

- P/Taint inherits all of the analysis support of Doop for complex Java language features (e.g., native methods, class loading, initialization, implicit reachability, and more). Notably, P/Taint transparently inherits Doop's sophisticated reflection analysis [Smaragdakis et al. 2015]: taint values can propagate through reflection operations, just as objects can. This enables information-flow analysis with high *recall*: we can detect many more information-flow violations in actual benchmarks, thus reducing *false negatives*.

We demonstrate the effectiveness of our approach on a number of benchmarks, including SecuriBench Micro [Livshits 2006], JInfoFlow-bench (a new benchmark suite we developed[2]), and DroidBench 2.0 [Arzt et al. 2014]. By leveraging the P/Taint's features, we achieve an over-99% recall rate over all Java information-flow benchmarks, with a very low incidence of false positives (over 91% precision). In order to further demonstrate the practical application of our approach, we P/Taint also supports many Android-specific features within its analyses. We have achieved a recall of over-96%, with over-86% precision for all relevant suites of DroidBench 2.0. The framework achieves such high-quality analysis results while maintaining very high efficiency—e.g., just over 7 minutes analysis time for all of SecuriBench. This efficiency is also complemented by high scalability—we apply P/Taint to real-world Android applications and show that we can Analyze, e.g., Facebook Messenger, an application with more than 55K classes, including the full Android library classes, in under seven hours.

2 BACKGROUND AND ILLUSTRATION

We next illustrate with simple examples the high-level ideas of our approach, while providing some background on points-to and information-flow analyses.

Points-To Analysis Basics. Points-to analysis computes the set of abstract objects that a program expression can refer to. Abstract objects are typically identified with allocation sites, i.e., instructions (instances of `new`) performing an object allocation: all run-time objects allocated by the same instruction are mapped into the same abstract object. (This is a standard simplification that ignores *context-sensitive heap abstractions*, a.k.a. *heap cloning*. We discuss this aspect separately in Section 4.3.)

The source of a points-to computation is, therefore, an allocation instruction:

```
A a1 = new A(); // heap allocation/abstract obj.
```

Points-to analysis will trivially infer that local variable `a1` can point to the abstract object identified with the above `new` statement. The essence of points-to analysis, however, is to compute how these abstract objects flow through different language constructs. The most interesting cases of standard constructs that propagate points-to information are method calls and heap loads/stores. For instance, our example can be extended with method calls and returns, as well as local assignments (possibly via casts) and heap loads and stores:

[2] Available online at https://github.com/plast-lab/JInfoFlow-bench .

```
Object p = foo();
bar(p);
B b1 = p.fld;
...
A foo() {
  A a1 = new A(); // abstract object A1
  return a1;
}
void bar(Object q) {
  A a2 = (A) q;
  a2.fld = new B(); // abstract object B1
}
```

Local variables `a1`, `p`, `q`, and `a2` can all point to (at least) abstract object `A1`. Similarly, local variable `b1` can point to abstract object `B1`. Abstract object `A1` (and, accordingly, any concrete object it represents) flows through the above program fragment via calls and returns. Abstract object `B1` flows through heap loads and stores.

The essence of a points-to analysis algorithm is in making these inferences as precisely as possible without missing any object flow. The end result of the analysis has a standard form of a map from a variable to a set of abstract objects (i.e., allocation sites).

Information-Flow Analysis Basics. Information-flow analysis computes which *sources* of data produce values that can reach specific data *sinks*. This general analysis statement captures several practical security and privacy questions: computing which tainted (i.e., untrusted) sources can provide values that are used in trusted computations; computing which privileged objects can flow to unprivileged code; computing which sensitive information can leak to unauthorized agents; etc.

The starting point of an information-flow computation is typically a data source API call. For instance, consider code that reads a string from an untrusted (*taint*) source, such as user input:

```
String a = source.readLine(); // taint source
```

The essence of information-flow is to track tainted values as they propagate through the program. The concept of taint is more abstract than objects, however. Taint is associated with the contents of the data and not with where they can be found in memory (or how they are allocated). For instance, two string objects that are allocated by the same instruction may have different taint values, depending on their contents.

Accordingly, the same taint can be passed across different object types. This is done via *data transform functions*. For instance, consider:

```
String a = source.readLine(); // taint source
byte[] aAsBytes = a.getBytes();
```

The `aAsBytes` variable holds a tainted value, with the same taint source as that of the `a` variable, even though the two are different objects, of incompatible types. The tainted string value was used to create a tainted byte array. However the contents have merely been transformed without losing the taint of the information.

Finally, taint can be removed via *sanitization* functions. These remove the taint from a value, in cases when the taint would otherwise propagate (e.g., when a new string contains parts of a tainted one). An example use of a sanitization function appears in the example below:

```
String a = source.readLine(); // taint source
String aSafe = java.net.URLEncoder.encode(a, "UTF-8");
```

Importantly, sanitization may apply even when the data storage (i.e., the object holding the information) remains the same: the exact same object will no longer have a tainted value.

Similarities and Differences. We observe from the above discussion that points-to analysis and information-flow analysis have similarities at a high level: both compute the flow of objects through a program. However, the two analyses also have significant differences:

- For points-to analysis, the values flowing to variables are abstract objects, i.e., allocation sites. For information-flow analysis, values identify taint sources, i.e., method calls.
- Taint and object identity are orthogonal concepts. The same taint can apply to different objects, and even objects of different type. The same abstract object (as far as points-to analysis is concerned) can represent both tainted and untainted concrete objects. The same object may be tainted or untainted at different points in its lifetime. Sanitization and data transform functions have no counterpart in the points-to analysis world.

These differences seem to make the two analyses fundamentally incompatible. Therefore, typical past implementations of a taint analysis (e.g., FlowDroid [Arzt et al. 2014]) have been clients of points-to analyses, using them to obtain a model of the heap. The taint analysis then defines on its own the flow of taint values in and out of heap objects, as well as through local variables.

Approach. Our approach consists of fully unifying points-to and information-flow analyses. We take standard points-to analysis algorithms and show how they can be modified to compute both points-to and information-flow results. The core of the resulting analysis remains the same: the flow of objects through methods or through the heap is determined by the original points-to analysis algorithm. However, a lot more values are now processed by this (virtually) unchanged logic: the key technique we employ is to identify information flow sources, as well as data transform functions, with artificial abstract objects, which the points-to analysis processes as if they were allocation sites.

To illustrate, consider our earlier example:

```
String a = source.readLine();      // artificial abstract object
byte[] aAsBytes = a.getBytes();    // parametric artificial abstract object
```

The unified analysis will treat the `readLine` method call as the source of an artificial abstract object—i.e., just like a `new` instruction for a points-to analysis. The artificial object represents a specific taint and it is passed around as a value, alongside regular objects. Abstract objects of the points-to analysis (i.e., regular allocation sites) and artificial abstract objects (i.e., taint source markers) are truly orthogonal—e.g., if a variable "points to" both kinds of objects, it is considered to receive tainted objects (with taint originating at the marked source) allocated at a given site.

Similarly, calls to data transform functions (such as `getBytes`, above) are also sources of artificial abstract objects. However, the call does not produce a single abstract object, but one per tainted value flowing into the call, which ensures that all the origins of tainted values are preserved.

In this setting, sanitization functions merely control the inter-procedural propagation of artificial abstract objects. The taint value simply does not flow into a sanitization method. In fact, this is the only small intervention that is required in the original algorithmic logic of a points-to analysis in order to support this approach: inter-procedural propagation needs to filter out artificial abstract objects, but not regular ones, when a matching sanitization method gets called.

Advantages. The unified approach offers significant advantages. Most importantly, information-flow analysis can transparently inherit all the functionality of a points-to analysis. This results in information-flow analyses of increased sophistication. The enhancement can affect both the precision and the recall of an analysis, i.e., its effectiveness in terms of both false positives and false negatives. For the former, we can employ significant precision enhancements, such as different flavors of context sensitivity [Milanova et al. 2002, 2005; Sharir and Pnueli 1981; Shivers 1991; Smaragdakis et al. 2011]. For the latter, we can use the detailed modeling of reflection that points-to analysis frameworks have developed to track the flow of values [Li et al. 2014, 2015b; Livshits 2006; Livshits et al. 2005; Smaragdakis et al. 2015]. We shall discuss these benefits in detail in Sections 4 and 7.

3 ANALYSIS DESIGN

We next demonstrate the unified points-to and information analysis approach in a model: a logic-based specification (in the syntax of the Datalog language, which is equivalent to first-order logic with recursion [Immerman 1999]) over a minimal input language. This serves two important purposes. First, it makes our description of the previous sections precise.[3] Second, it showcases the generality of the approach. The model we present is an incremental addition over a standard logical specification, for Andersen-style points-to analyses. Smaragdakis and Balatsouras [2015] show that this model captures virtually all realistic pointer analysis algorithms and can be enriched with many features, such as context-sensitivity, arrays, reflection analysis, exception-flow analysis, and more. These additions are transparent relative to our minimal changes to the base model. This allows information-flow analysis to be unified with a large variety of points-to analyses.

3.1 From Pointer to Information-Flow Analysis

We begin by discussing the preliminary changes to enable the analysis of information flow simultaneously with points-to analysis, ignoring data transform functions and sanitization. We add support for these features in Sections 3.2 and 3.3.

Fig. 1 shows the domain of our analysis, its inputs, computed relations and outputs, as well as a constructor function that produces new tainted values. Fig. 2 shows the analysis. We enclose in red boxes the elements that have been modified to support information flow, unless they are mere renamings of arguments and predicates. The rule syntax is simple: the left arrow symbol (←) separates the inferred facts (i.e., the head of the rule) from the previously established facts (i.e., the body of the rule).

We explain the contents and changes in both figures in more detail below.

Schema Changes. We define a value abstraction A that encompasses both heap abstractions, A_H, (defined as object allocation sites in pointer analysis) and tainted value abstractions, A_T. As we saw in the previous section, tainted value abstractions are similar to heap abstractions, and also represent sites in the program where the values are introduced. We rename the main output relations of points-to analysis, VarPointsTo and FldPointsTo, to FlowsToVar and FlowsToFld, respectively, to better match their new roles. The output or intermediate computed relations (Leak, FlowsToVar, FlowsToFld) are defined in terms of value abstractions, where previously they were defined on heap abstractions. In almost all cases both heap and tainted value abstractions are treated identically in our rules.

Algorithmic Changes. The striking feature of Fig. 2 is the lack of changes in the main rules that propagate values through the heap and through method calls. The first eight rules in the Figure are the standard model of a (context-insensitive) points-to analysis. The only change is in the computation of a call graph (predicate CallGraph). Not all value abstractions are used when computing receiver objects for virtual calls: the receiver object has to be a regular heap abstraction. This check prevents the analysis from reaching methods that are not truly reachable for any real object, even though an

[3] In a close analogy, the latest edition of the Java Virtual Machine specification [Lindholm et al. 2015, p.170-320] offers a Prolog/Datalog specification of the JVM verifier, using *"English language text [...] to describe the type rules in an informal way, while the Prolog clauses provide a formal specification."*

V is a set of program variables
A_H is a set of heap value abstractions (i.e., allocation sites)
$\boxed{A_T \text{ is a set of tainted value abstractions } (A_H \cap A_T = \emptyset)}$
$\boxed{A \text{ is a set of all value abstractions (i.e. } A_H \cup A_T)}$
M is a set of method identifiers
S is a set of method signatures (including name, type signature)
F is a set of fields
I is a set of instructions
T is a set of class types
$\mathbb{N}$ is the set of natural numbers

$\textsc{Alloc}(var : V, heap : A_H, meth : M)$	# *var = new ...*
$\textsc{Move}(to : V, from : V)$	# *to = from*
$\textsc{Load}(to : V, base : V, fld : F)$	# *to = base.fld*
$\textsc{Store}(base : V, fld : F, from : V)$	# *base.fld = from*
$\textsc{VCall}(base : V, sig : S, invo : I, meth : M)$	# *base.sig(..)*
$\boxed{\textsc{SourceMethod}(meth : M, type : T)}$	
$\boxed{\textsc{SinkMethod}(meth : M, n : \mathbb{N})}$	

$\textsc{FormalArg}(meth : M, n : \mathbb{N}, arg : V)$
$\textsc{ActualArg}(invo : I, n : \mathbb{N}, arg : V)$
$\textsc{FormalReturn}(meth : M, ret : V)$
$\textsc{ActualReturn}(invo : I, var : V)$
$\textsc{ThisVar}(meth : M, this : V)$
$\textsc{ValueType}(value : A, type : T)$
$\textsc{LookUp}(type : T, sig : S, meth : M)$

$\boxed{\textsc{Leak}(value : A_T, sink : I)}$
$\textsc{FlowsToVar}(var : V, value : A)$
$\textsc{CallGraph}(invo : I, meth : M)$
$\textsc{FlowsToFld}(baseH : A, fld : F, value : A)$
$\textsc{InterProcAssign}(to : V, from : V)$
$\textsc{Reachable}(meth : M)$

$\boxed{\textbf{NewTaintedValue}(invo : I, type\!: T) = value : A_T}$

Fig. 1. Our domain, input relations, computed relations and constructor of tainted values, adapted from Smaragdakis and Balatsouras [2015]. The input relations are of two kinds: relations encoding program instructions and relations encoding type system and other environment information. New additions are enclosed in red boxes.

artificial taint object appears to reach the invocation site. This can arise because of slightly different propagation rules for heap abstractions and taint value abstractions—for instance, taint transform rules (Section 3.2) introduce shortcuts that may facilitate the flow of taint values even in cases where normal objects would not propagate.

The last two rules are new. They simply mark calls to information sources as sites where abstract values are created, and information sinks as sites of leaks, to be reported as part of the output.

The first of the two rules states that every time a source method (input predicate SOURCEMETHOD) is computed to be reachable (using CALLGRAPH), the variable receiving the return value (*to*) points to a new abstract value. The introduction of new abstract values is done via the constructor

FlowsToVar(*var, value*) ←
  Reachable(*meth*),
  Alloc(*var, value, meth*).

FlowsToVar(*to, value*) ←
  Move(*to, from*),
  FlowsToVar(*from, value*).

Reachable(*toMeth*),
FlowsToVar(*this, value*),
CallGraph(*invo, toMeth*) ←
  VCall(*base, sig, invo, inMeth*),
  Reachable(*inMeth*),
  FlowsToVar(*base, value*), $\boxed{value \in A_H}$,
  ValueType(*value, heapT*),
  Lookup(*heapT, sig, toMeth*),
  ThisVar(*toMeth, this*).

InterProcAssign(*to, from*) ←
  CallGraph(*invo, meth*),
  FormalArg(*meth, n, to*),
  ActualArg(*invo, n, from*).

InterProcAssign(*to, from*) ←
  CallGraph(*invo, meth*),
  FormalReturn(*meth, from*),
  ActualReturn(*invo, to*).

FlowsToVar(*to, value*) ←
  InterProcAssign(*to, from*),
  FlowsToVar(*from, value*).

FlowsToFld(*baseH, fld, value*) ←
  Store(*base, fld, from*),
  FlowsToVar(*from, value*),
  FlowsToVar(*base, baseH*).

FlowsToVar(*to, value*) ←
  Load(*to, base, fld*),
  FlowsToVar(*base, baseH*),
  FlowsToFld(*baseH, fld, value*).

NewTaintedValue(*invo, type*) = *value*,
FlowsToVar(*to, value*) ←
  SourceMethod(*meth, type*),
  CallGraph(*invo, meth*),
  ActualReturn(*invo, to*).

Leak(*value, sink*) ←
  CallGraph(*sink, meth*),
  SinkMethod(*meth, i*),
  ActualArg(*sink, i, from*),
  FlowsToVar(*from, value*), $value \in A_T$.

Fig. 2. Datalog rules for points-to analysis, call-graph construction and simple information-flow analysis.

NewTaintedValue. This constructor function is a parameterization point for the analysis, but it will typically be defined to retain all information passed to it:

NewTaintedValue(*invo, type*) = ***pair(invo, type)*** : A_T

That is, new value abstractions will encode both the invocation site and the value type. Taint can be transferred from one value abstraction to another, and we will see later how this encoding helps ensure that value abstractions only get a single type. The invocation site is useful in order to record where the tainted value is originally introduced in the program.

The last rule is a mere consumer of the analysis results. It states that a leak is found if an abstract taint value reaches the i-th argument at the invocation site (*sink*) of a sink method whose i-th argument is sensitive (SinkMethod(*meth,i*)).

3.2 Flow-Through Data Transform Functions

As we saw in Section 2, taint can flow through data transform functions, so that values of different types become tainted. To support data transform functions, we augment our earlier model with the extra relations and rules in Fig. 3. These rules add extra logic for the creation of taint objects—the original rules of Fig. 2 remain unchanged and values flow identically through the heap, local variables,

```
BaseToRetTransfer(meth : M, type : T)
ArgToRetTransfer(meth : M, n : ℕ, type : T)
------------------------------------------------
IsTaintedFrom(to : V, from : V, type : T)
------------------------------------------------
TransferTaint(value : A, type : T) = newValue : A_T
------------------------------------------------

IsTaintedFrom(from, to, type) ←
  CallGraph(invo, meth),
  BaseToRetTransfer(method, type),
  VCall(from, _, invo, _),
  ActualReturn(meth, to).

IsTaintedFrom(from, to, type) ←
  CallGraph(invo, meth),
  ArgToRetTransfer(method, n, type),
  ActualArg(invo, n, from),
  ActualReturn(meth, to).

TransferTaint(value, type) = newValue,
FlowsToVar(var, newValue) ←
  IsTaintedFrom(from, to, type),
  FlowsToVar(from, value), value ∈ A_T,
  FlowsToVar(to, oldValue),
  Alloc(var, oldValue, _).
```

Fig. 3. Transferring information: two new input predicates, a new intermediate predicate, a new constructor, and rules that transfer taint through predefined methods.

and method calls. This is a representative but not complete list of taint transfer rules. P/Taint defines more ways in which taint can be transferred. For instance, a method may be tainting only a specific field of a returned object.

Fig. 3 defines two new input relations, BaseToRetTransfer and ArgToRetTransfer, an intermediate predicate IsTaintedFrom, and new constructor function **TransferTaint**. The new input relations encode how the return value of a (data transform) method is tainted, as a function of either the method call's receiver variable (base variable) or argument. For convenience, the new tainted value's type is listed explicitly. The first two rules of Fig. 3 combine the input information into a more general intermediate predicate, IsTaintedFrom(*from,to,type*), capturing which variables hold the input and the new tainted value, as well as the type of the latter.

The third rule of the figure is responsible for creating a new tainted value when an existing tainted value flows to a transform function. The rule body contains a subtle feature: the new value is created not at the point of return of the transform function (variable *to*) but at any point where heap objects are allocated and assigned for the first time, as long as these objects flow to *to*. We have found this to yield maximum generality and precision in our P/Taint implementation.

The new tainted value is created using the **TransferTaint** constructor, employed at the head of the rule. The constructor is a parameter to the analysis model: different definitions will determine how many taint abstract objects are created, affecting precision and scalability. A reasonable definition is:

TransferTaint(*value, type*) = ***pair(first(value), type)*** $: A_T$

$$\begin{aligned}&\textsc{FlowsToVar}(\mathit{to, value}) \leftarrow\\ &\quad \textsc{InterProcAssign}(\mathit{to, from}),\\ &\quad \textsc{FlowsToVar}(\mathit{from, value}),\ \boxed{\mathit{value} \in A_H}.\end{aligned}$$

$\textsc{SanitizationMethod}(\mathit{meth} : M)$

$$\begin{aligned}&\textsc{FlowsToVar}(\mathit{to, value}) \leftarrow\\ &\quad \textsc{InterProcAssign}(\mathit{to, from}),\\ &\quad \textsc{FlowsToVar}(\mathit{from, value}),\ \mathit{value} \in A_T,\\ &\quad !\ (\textsc{FormalArg}(\mathit{meth}, \_, \mathit{to}),\\ &\quad \textsc{SanitizationMethod}(\mathit{meth})\ ).\end{aligned}$$

Fig. 4. Sanitization: We define a new input relation SanitizedMethod and modify the inter-procedural assignment logic defined in Fig. 2.

That is, the new tainted value encodes the first half of the input tainted value (i.e., the invocation site of the original taint source) and the type of the new tainted value. The new taint value has the appropriate type and propagates through the rest of the analysis without thwarting type invariants. Precise type information for value abstractions is used, for instance, to limit the propagation of values through casts, which results in a more precise analysis.

In practice, this encoding not only ensures that the number of tainted values remains finite, but aids in performance. An original value's taint is typically transferred several times during an analysis. It is not necessary to construct a new tainted value each time if a tainted value of the same type exists that originates from the same invocation site.

3.3 Sanitization Functions

The final element that information-flow analysis needs to support is sanitization of values. Sanitization simply disallows the flow of tainted values through predefined functions. In order to support this functionality, we introduce a new input relation, SanitizationMethod, and modify the inter-procedural assignment logic. Fig. 4 shows the result. Note that the last rule modifies the corresponding rule in Fig. 2, while the first rule is new.

The rules handle heap abstract objects differently from taint abstract objects. The former propagate as in the original analysis. The latter propagate only if it is not the case that the target of the assignment (variable *to*) is a formal argument of a sanitization method.

4 SCALING TO A FULL TAINT ANALYSIS FRAMEWORK: BENEFITS

The model of the previous section is a good illustration of the principles behind the unified analysis approach. It is also the basis of our realistic information-flow analysis framework, P/Taint. P/Taint is built on top of Doop [Bravenboer and Smaragdakis 2009b]—a full-fledged points-to analysis library. Thanks to the seamless unification of pointer and information-flow analysis, a large part of the features and algorithmic enhancements of P/Taint are inherited directly from Doop. Indeed, P/Taint is now fully integrated with the standard Doop framework [4]. We proceed to describe how the approach scales to a realistic language and what benefits we get from it.

4.1 Broad Support of Java Semantics

The minimal model of the previous section ignores several realistic semantic complications. These include support for the full Java bytecode language (including: static fields and methods; casts; arrays; exception handling; finalization; class initialization; final fields), as well as semantics prescribed

[4] Available online at: https://bitbucket.org/yanniss/doop

in the Java Virtual Machine specification (including: JVM initialization; initial threads and thread groups; privileged actions). These features need to be modeled for a security analysis that has a realistic claim to completeness. Furthermore, modeling these features has a far-from-trivial effect on static analysis complexity. For instance, handling exceptions more precisely yields an order-of-magnitude performance improvement for many analyses [Bravenboer and Smaragdakis 2009a]; modeling privileged actions (i.e., objects handled to the JDK's `doPrivileged` method) is crucial for analyzing realistic programs and their inclusion often results in a 2-orders-of-magnitude slowdown.

A large part of the benefit of the unified points-to/information-flow approach is that the above semantic complications are only handled once. The implementation of P/Taint transparently includes the corresponding analysis features from Doop, with no further modification necessary. Essentially, all modeling of language semantics concerns the flow of values and remains unchanged regardless of whether these values are abstract heap objects or abstract taint objects.

Importantly, we cannot get the same benefit by having information-flow analysis be a mere client of points-to analysis, i.e., consuming its results only. For instance, it is not enough to know that an abstract (heap) object was originally tainted and flows to a sink. We also need to track its taint separately (through the full set of language features) and see if it also flows to the same sink or gets filtered out by a sanitization function. Similar complications arise in the case of data transform functions: we need to know the exact kind of taint (i.e., its source) that flows to such a function, through any language mechanism (e.g., exceptions or implicit initialization) to determine the tainted value produced.

4.2 Reflection

One language feature that is worth special attention is reflection. Reflection allows highly dynamic behavior in an otherwise static language: reflection operations can be used to create objects of a dynamically-determined type and access dynamically-determined methods or fields of any object. Handling reflection in static analysis is a complicated topic with an ever-growing literature [Li et al. 2014, 2015b; Livshits et al. 2005; Smaragdakis et al. 2015]. Furthermore, handling reflection is crucial for security analyses [Lerch et al. 2014; Livshits 2006].

P/Taint inherits the Doop support for reflection, which significantly enhances the framework's ability to detect information-flow violations. Taint values are propagated through reflection operations transparently, much like they are through the heap and calling stack in our earlier analysis rules. Reflection support constitutes about one-fifth of the Doop core code (i.e., the code excluding different points-to analysis variations): over 2KLoC, or some-200 logical rules. Replicating this machinery would have required significant effort.

4.3 Context Sensitivity and Precision

An important aspect of every static analysis is precision. In the case of security analyses, this translates to a low false-positive rate for analysis warnings. A major way to gain precision for a points-to analysis is via *context sensitivity*: qualifying variables and abstract objects with context information, so that two instances of the same local variables (for different invocations of the method) or of the same abstract object are not conflated.

Acquiring precision through context sensitivity is one of the benefits of the unified points-to/information-flow analysis approach. The model of Section 3 is context-insensitive, yet there is a well-established pattern for adding context sensitivity to the same rules [Smaragdakis and Balatsouras 2015]. Furthermore, the pattern allows different context models to be plugged in, by defining constructors **Record** and **Merge** [Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2011]. (Indeed, the Doop framework underlying P/Taint is the practical incarnation of this model of parametric context sensitivity.) We can, thus, exploit *call-site sensitivity* [Sharir and Pnueli 1981;

$$\begin{array}{l}
\textbf{Merge}(\mathit{value, hctx, invo, callerCtx}) = \mathit{calleeCtx}, \\
\textsc{Reachable}(\mathit{calleeCtx, toMeth}), \\
\textsc{FlowsToVar}(\mathit{calleeCtx, this, hctx, value}), \\
\textsc{CallGraph}(\mathit{callerCtx, invo, calleeCtx, toMeth}) \leftarrow \\
\quad \textsc{VCall}(\mathit{base, sig, invo, inMeth}), \\
\quad \textsc{Reachable}(\mathit{callerCtx, inMeth}), \\
\quad \textsc{FlowsToVar}(\mathit{base, value}),\ \mathit{value} \in A_H, \\
\quad \textsc{ValueType}(\mathit{value, heapT}), \\
\quad \textsc{Lookup}(\mathit{heapT, sig, toMeth}), \\
\quad \textsc{ThisVar}(\mathit{toMeth, this}).
\end{array}$$

Fig. 5. Context-sensitive version of earlier rule. (**Merge** is the standard constructor for new contexts at virtual call sites.)

Shivers 1991], *object sensitivity* [Milanova et al. 2002, 2005], *type sensitivity* [Smaragdakis et al. 2011], and *hybrid sensitivities* [Kastrinis and Smaragdakis 2013].

In the full, context-sensitive, versions of the rules of Section 3, *ctx* and *hctx* parameters (for variable and object contexts, respectively) are added to all rules. For instance, the rule inferring CallGraphEdge in Fig. 2 is shown in context-sensitive form in Fig. 5. **Merge** is used to create new calling contexts (or just "contexts") at virtual call sites. These contexts are used to qualify method calls, i.e., they are applied to all local variables in a method. The **Merge** function takes all available information at the call-site of a virtual method call and combines it to create a new context (if one for the same combination of parameters does not already exist). Different definitions of **Merge** yield different context-sensitivity flavors.

4.4 Other Pragmatic Features

Pragmatic features that aid in the practicality of P/Taint can naturally fit within its simple design. One of these is the ability to label sources and sinks, a feature that enables the checking of simple security policies. Labeling is implemented by augmenting the **NewTaintedValue** construct, primarily by adding an argument containing the label of the source. The construct indexes the tainted value's origin with the label so that the label can be retrieved when FlowsTo data is queried. Labels add a negligible constant overhead to the analysis. Furthermore, one can check simple security policies. For example, checking if personally identifiable information is flowing to insecure public channels, or filtering out from the results medium sensitivity information flowing into logs.

Moreover, P/Taint supports *breadcrumbs*, which can be associated with tainted value abstractions to help the framework user debug the information flow in her application. Breadcrumbs enable the framework user to understand not just the provenance of the information but also the intermediate steps in its flow. These pieces of information can be added to the tainted value abstractions by taint transfer rules and methods. The **TransferTaint** construct takes a selection of predefined breadcrumbs and the information is preserved with the information-value abstraction each time the taint is transferred from one value abstraction to another. Breadcrumbs can be post-processed and presented in queries to help piece together the flows of information.

4.5 Overall Benefit

We implemented the core information-flow analysis logic of P/Taint on top of the Doop framework in just ~200 lines of Datalog code.[5] Apart from renamings and simple refactorings, only ~30 lines of the Doop code were modified. These are well-captured by the model of Section 3: they treat taint abstract objects differently for purposes of sanitization and call-graph construction.

Indeed, much of the effort of designing and implementing the P/Taint information-flow analysis consisted of non-coding or orthogonal tasks:

- configuring sources, sinks, and sanitizers using regular expressions that encode method descriptors;
- checking that the approach indeed works without needing changes to existing points-to analysis code—i.e., that taint abstract objects should indeed be propagated unchanged through all the handling of Java semantics.

The experience has been a striking (to us) validation of the value of integrating points-to and information-flow analysis.

5 DISCUSSION

We next discuss further the advantages and disadvantages of the P/Taint approach, contrasting it with alternatives.

5.1 Contrast with Conventional Approaches

To see the key features of the unified analysis, we can consider it in comparison to analysis approaches that have largely similar purposes, yet a conventional structure.

Conventional Datalog-based taint analysis. The Beacon static analysis tool [Karim et al. 2012] aims to detect capability leaks in Mozilla Jetpack modules. Beacon, much like P/Taint, is performing a taint analysis using Datalog inference rules. Although the use of Datalog is orthogonal to the principles of our joint points-to/information-flow approach, it is helpful to have a succinct encoding of the two analyses so that their differences are clear.

In Beacon, taint analysis is a client of points-to analysis. There are separate relations PTSTO(*var, val*) and IDISTAINTED(*var, taintType*), both of which correspond to our FLOWSTOVAR(*var, val*). Similarly, Beacon has separate relations HEAPPTSTO(*obj, fld, val*) and ISTAINTED(*obj, taintType*), both of which correspond to our FLOWSTOFLD(*obj, fld, val*). This results in duplication of effort—e.g., there are two slightly different recursive rules [Karim et al. 2012, Table 6] that form the basis of the analysis by combining the main relations: one rule joins PTSTO and ISTAINTED to produce IDISTAINTED, while the other joins PTSTO and HEAPPTSTO to produce PTSTO. Such duplication percolates to all language features of a realistic analysis.

A feature of P/Taint's unified approach is the representation of taint as new abstract values that propagate independently of regular objects (through the exact same rules for value flow). In contrast, in Beacon, object allocations are tagged with taint values, employing a separate input relation ISPRIVILEGED(*obj, taintType*). This design decision goes hand-in-hand with the choice to have taint analysis as a client (i.e., consumer of results) of a plain, unenhanced points-to analysis. By tagging heap objects at their allocation point as tainted, points-to analysis can merely propagate the objects (just as it would for an analysis without taint considerations). Taint analysis can then watch where objects have propagated and propagate taint maximally. In Beacon, this means that if an object merely

[5] The framework is much larger, but due to orthogonal functionality, aimed at security analysis, such as support for Android or for analyzing open programs (discussed in Section 6),

refers to a tainted object (even without reading values from tainted fields), it is considered tainted. Comparing the two designs showcases the benefits of the P/Taint approach on multiple dimensions:

- The P/Taint approach avoids the need to tag heap allocations with taint labels. This is low-level information that the user may not have, and the allocations themselves may even be unavailable for labeling (e.g., objects may be returned by native code). Labeling source methods instead is more convenient and general.
- The P/Taint approach enables support for sanitization: a regular object can propagate even when an associated taint value does not. In Beacon, tagging with a taint is a property of a heap object but the heap object may have propagated to several points in the program before being tainted, and can propagate to many others after being sanitized. Thus, no distinction of tainted/untainted versions can be made.
- The P/Taint approach enables labeling primitive values, not merely heap objects. As a concrete example, a method such as "`int BufferedReader.read()`" is merely labeled as an information source. In this way, it implicitly creates taint values, even though it does not create objects (since it returns `int`s). These taint values propagate throughout the value-flow analysis of all language features with no extra effort needed.
- P/Taint's analysis is fundamentally more precise. If an object is tainted and there is a reference to it from some other object, the parent object is never tainted by default. This makes many fewer objects tainted, decreasing the false-positive rate. E.g., P/Taint's analysis is able to distinguish whether the head or one of the tail objects of a linked list is tainted.

In all, the unified P/Taint approach yields a full-fledged information-flow analysis, complete with sanitization support, benefiting from a deep reuse of all the sophistication of points-to analysis.

Security analysis for frameworks. The unified P/Taint approach leverages, without extra effort, a general-purpose points-to analysis of the entire program, including libraries. Thus, compared to conventional approaches, P/Taint has less of a need for ad hoc models of environment-specific behavior. For instance, many taint analysis frameworks for Android (e.g., FlowDroid [Arzt et al. 2014] or DroidSafe [Gordon et al. 2015]) do not analyze the underlying library. Library operations, e.g., `java.util.HashSet.add` are instead explicitly modeled as taint transfer functions, so that if a tainted object is added to a set, the underlying `Set` is also tainted. Although it is tempting to shortcut the analysis by defining taint transfer functions specific to the domain, this will always be an incomplete exercise.

P/Taint does not need definitions like these. Instead the underlying points-to analysis models the flow through the low-level data structures. The taint travels from a `java.util.HashSet` to an underlying `java.util.HashMap`, to the underlying `java.util.HashMap.Entry`, etc. This generic approach to taint transfer is very effective, especially when analyzing applications in the wild. Large software houses tend to use a third party data structures, sometimes even defining their own for various reasons instead of relying on what is offered in the collections library.

5.2 Limitations

A unified points-to/taint analysis inherits the benefits of mature points-to analysis frameworks, yet is also limited by what the framework can express. This is well illustrated both in our analysis model and in the full P/Taint implementation. Although the model of Section 3 captures a wealth of realistic points-to analysis algorithms, it also fixes important design parameters, thus constraining the analyses expressible in it. Two such limitations are notable:

- The model (as well as the Doop framework underlying P/Taint) captures a *flow-insensitive* analysis. The analysis builds a heap abstraction that does not vary per-program-point. (To a lesser extent, this limitation also applies to local variables, however this is addressed by a static-single-assignment pre-processing of the input.) Past information-flow analyses—e.g., FlowDroid—have integrated partial flow sensitivity to increase analysis precision. Some code patterns will likely require a flow-sensitive treatment for full precision. However, context sensitivity is generally a better trade-off for pointer-analysis precision and performance. In our experiments we have found that a context-sensitive analysis yields excellent precision in all but very few cases.

- The analysis model does not capture analyses that use an *access-path-based* formulation. Access paths are expressions of the form "*var*(*.fld*)*", i.e., field-access expressions of any length. A pointer analysis can employ access paths in concepts such as "this access path points to this value" or "these access paths are aliased". This can be extended to taint analysis—e.g., the Andromeda tool [Tripp et al. 2013] computes tainted and aliased access paths in a dual data-flow analysis. Access paths are generally advantageous for modular analyses, e.g., to avoid computing a whole-program image of the heap.

Both of these limitations concern the current formulation of unified points-to and taint analysis, and not the general idea. In principle, a flow-sensitive points-to analysis or an access-path-based analysis can be used as bases for unification with an information-flow analysis. Such unification can employ the same key elements as in our formulation: taint can be represented as abstract values that flow alongside regular analysis values.

6 PRACTICAL ELEMENTS

No practical analysis framework can yield realistic results without engineering effort to support the idiosyncrasies of different environments. We next discuss the P/Taint support for such important practicalities.

6.1 Android Support

P/Taint supports Android, with a feature set very similar to FlowDroid [Arzt et al. 2014]. A list of sources and sinks for Android was curated and labeled according to predefined criteria (e.g., Telephony, Logging, Net). We made several more enhancements of general value, largely orthogonal to taint analysis. For instance, the underlying Doop analysis was enhanced with support for analyzing Android apps, in the form of identifying entry points and UI elements described in XML configuration files (a.k.a. the application's "manifest").

Android lifecycle. P/Taint (and the underlying Doop framework) models the Android component lifecycle. It reads the manifest in an application's APK file to discern the application's components. Android components include activities, services, broadcast receivers, etc. and can have multiple entry points. GUI widgets are listed in the manifest and can be linked to their respective application components. Components also register for callbacks on events. In addition to making the callback reachable, the caller (the component raising the event) needs to be linked to the callee.

Inter-component communication. Inter-component communication is naively modeled, in a way similar to FlowDroid's modeling of inter-component communication. P/Taint over-approximates explicit inter-component communication, for instance by modeling methods that send explicit messages ("intents") as sinks and callbacks that receive intents as sources.

Misc. Android patterns. Further improvements to the underlying analysis were performed in order to support the idiosyncrasies of Android applications. Tainted values can also emanate from GUI components. For instance, password components that are specifically tagged in the manifest as password fields are treated specially: any textual input read from these fields is flagged as tainted. Furthermore, Dalvik bytecode does not have instructions to create multi-dimensional arrays. Instead, it uses the reflection API to construct such arrays. Although P/Taint has generic reflection support, we have optimized this specific pattern for performance and precision as this is a regularly occurring pattern.

6.2 Conventional Java Program Support

Conventional JVM programs also need specific analysis support for common Java features. A common case of Java programs is servlet applications, meant to be executed inside a web server. P/Taint supports them via features for analyzing *open programs*, i.e., programs with multiple public entry points and not a single `main` method to start the computation. The analysis only requires an accurate model of the environment in order to compute a precise call graph. P/Taint's support for open programs finds entry points for servlets and adds them to the call graph. It also instantiates the environment of servlet applications, i.e., receiver objects, formal parameters to entry points, etc.

We introduce some level of generic handling of many string operations by modeling the internals of the `java.lang.String` class. In order to model taint transfer through various kinds of string operations, we model String objects so that their taint is derived from their internal `char[] value` field. This means that if a `String` object is tainted, its internal char array is treated as tainted and vice versa. The advantage of this approach is that it adds generality. For instance, a user may introduce new operations (e.g., writes his/her own version of concat), or operations may be added to the `String` class in other versions of the JRE. This approach only works for pure Java operations, so common string operations which rely on native code under the hood (e.g., `StringBuilder` operations) are modeled explicitly.

Serialization and deserialization of objects containing tainted values is also modeled explicitly, as serialization and deserialization is performed using native code.

7 EVALUATION

In this section, we present the results of our experimental evaluation of P/Taint. There are several research questions that our experiments intend to answer:

RQ.A Does our unified points-to/information-flow analysis approach yield *precision* benefits, i.e., a low rate of false positives?

RQ.B Does the unified points-to/information-flow analysis approach yield *recall* benefits, i.e., a low rate of false negatives?

RQ.C Is P/Taint overall effective at detecting information flow violations with high precision, compared to other tools in the literature?

RQ.D Is P/Taint scalable and is it efficient in terms of run time?

We have two distinct experimental setups. One consists of a set of controlled benchmarks: suites with ground truth (in terms of labeled information leaks) and source code we can inspect. The other is a set of large "applications in the wild": Android apps of substantial size without labeled information flow sources/sinks/violations.

We ran P/Taint on an idle machine with an Intel Xeon E5-2687W v4 3.00GHz and 256GB of RAM. We used the PA-Datalog engine, a publicly available, stripped-down version of the commercial LogicBlox Datalog engine. In addition to the LogicBlox version evaluated here, we are also releasing a newer version of P/Taint that uses the Soufflé [Jordan et al. 2016] Datalog engine. This can run more than twice as fast on multicore machines on many benchmarks.

7.1 Controlled Benchmarks

We use several controlled benchmarks to evaluate P/Taint:

SecuriBench Micro. [Livshits 2006] This benchmark contains 122 labeled servlets with possible security flaws (e.g., SQL injection).

JInfoFlow-bench. 12 plain Java benchmarks exercising reflection, event-driven architecture, and popular software engineering patterns.

DroidBench 2.0. [Arzt et al. 2014] Multiple Android security benchmark suites.

SecuriBench Micro benchmarks are servlet applications. P/Taint's open program support (Section 6) eliminates the need to write a test harness for each individual SecuriBench application. Using this feature we were able to analyze the entire benchmark suite (all 122 servlets) simultaneously, producing an application with 266 entry points and 3.2K classes, including libraries. While running the benchmarks, we inspected SecuriBench Micro manually and (on very few instances) corrected the reported numbers relative to the official benchmark page.[6] (For instance, test case Basic31 has meta-data that claim two vulnerabilities, while the code clearly documents three. This makes the total number of vulnerabilities in the basic package be 61 instead of 60.)

JInfoFlow-bench is a benchmark suite that we developed to distill some popular but hard-to-analyze software engineering patterns. The programs in the suite require a highly precise analysis, as well as reflection support to detect information-flow violations. For instance, one benchmark encodes an event framework, which relies heavily on reflection. JInfoFlow-bench is platform- and library-agnostic. Since the benchmark suite does not rely on the use of external libraries for most of its behavior, it should be harder to write a security analyzer specifically optimized for this benchmark.

DroidBench 2.0 is a collection of benchmark suites designed to test for various vulnerabilities in Android applications. It contains 118 distinct Android applications spread over several benchmark suites. We evaluate P/Taint on all these benchmarks, except for benchmarks within the *implicit-flow* suite (4 applications) and *emulator detection* (3 applications). The latter benchmark suite is intended for dynamic analysis, to detect whether the application behaves differently under emulation, while the former is intended for taint analyzers that model control dependencies.

We analyzed SecuriBench Micro and JInfoFlow-bench with JRE 7 and Servlet API 3.0, and analyzed DroidBench with Android 25.

7.1.1 Overall Effectiveness. As discussed earlier, building P/Taint on top of the Doop framework gives us the ability to employ highly precise analyses, due to context-sensitivity, and thorough Java language feature support (including reflection).

Figures 6 and 7 make this argument concrete. The figures show a summary of the results of running P/Taint, using a highly precise form of context sensitivity (selective hybrid 2-object-sensitive+heap [Kastrinis and Smaragdakis 2013]) with reflection analysis support, on all JVM and Android controlled benchmarks.

The overall result is very high recall (**RQ.B**) and very high precision (**RQ.A**). For instance, in the JVM controlled benchmarks the analysis achieves 91% precision and 99% recall in SecuriBench Micro, as well as 100% precision and recall in our own JInfoFlow benchmark. The analysis issues just 13 false positives for over 150 detected information-flow violations in these benchmarks. For most benchmark sub-suites, P/Taint yields a perfect score for precision and recall. The false positives that were recorded are mostly due to the fact that P/Taint is not flow- or path-sensitive, nor is it array-sensitive, i.e., it does not discriminate a specific array index from another.

P/Taint fares almost equally well on Android benchmarks (DroidBench), with 86% precision and 96% recall, although supporting Android had not been the main focus of the work. A similar pattern as before emerges: P/Taint's false positives are mainly due to it not being flow-sensitive. (In fact, the suite FieldAndObjectSensitivity mostly tests for flow sensitivity rather than object sensitivity.)

[6] https://suif.stanford.edu/~livshits/work/securibench-micro/descr.html

Suite	TP	FP	FN	Precision	Recall
Total	139	13	1	**91%**	**99%**
aliasing	12	0	0	100%	100%
arrays	9	5	0	64%	100%
basic	61	0	0	100%	100%
collections	14	2	1	88%	93%
datastructures	6	0	0	100%	100%
factories	3	0	0	100%	100%
inter	17	0	0	100%	100%
pred	5	3	0	62%	100%
reflection	4	0	0	100%	100%
sanitizers	4	0	0	100%	100%
session	3	1	0	75%	100%
strong updates	1	2	0	33%	100%
Total	15	0	0	**100%**	**100%**
JInfoFlow/basic	2	0	0	100%	100%
JInfoFLow/ctx	5	0	0	100%	100%
JInfoFlow/event	4	0	0	100%	100%

Fig. 6. Summary of results for all JVM controlled benchmarks analyzed under selective hybrid 2-object sensitivity + heap context sensitivity and reflection support enabled. The top group of benchmarks breaks down the 12 sub-suites of SecuriBench, while the bottom comprises the 3 sub-suites of JInfoFlow-bench.

Suite	Vulnerabilities	TP	FP	FN	Precision	Recall
Total	99	95	16	4	**86%**	**96%**
Aliasing	0	0	1	0	0%	
AndroidSpecific	11	9	2	2	82%	82%
ArraysAndLists	3	3	4	0	43%	100%
Callbacks	16	16	4	0	80%	100%
FieldAndObjectSensitivity	2	2	2	0	50%	100%
GeneralJava	21	21	0	0	100%	100%
InterAppCommunication	2	2	2	0	50%	100%
InterComponentCommunication	18	17	1	1	94%	94%
Lifecycle	17	16	0	1	100%	94%
Reflection	4	4	0	0	100%	100%
Threading	5	5	0	0	100%	100%

Fig. 7. Summary of results for all relevant DroidBench suites. The analysis is identical to Figure 6.

It is worth noting that P/Taint achieves highly competitive precision and recall relative to those reported by recent state-of-the-art systems (**RQ.C**) like FlowDroid [Arzt et al. 2014] and Pidgin [Johnson et al. 2015]. Pidgin achieves 90% precision and 97% recall on the SecuriBench Micro benchmark suite, per its published numbers. FlowDroid is reported to achieve 93% precision and 97% recall on a subset of SecuriBench Micro (that does not include the “reflection”, “sanitizers”,

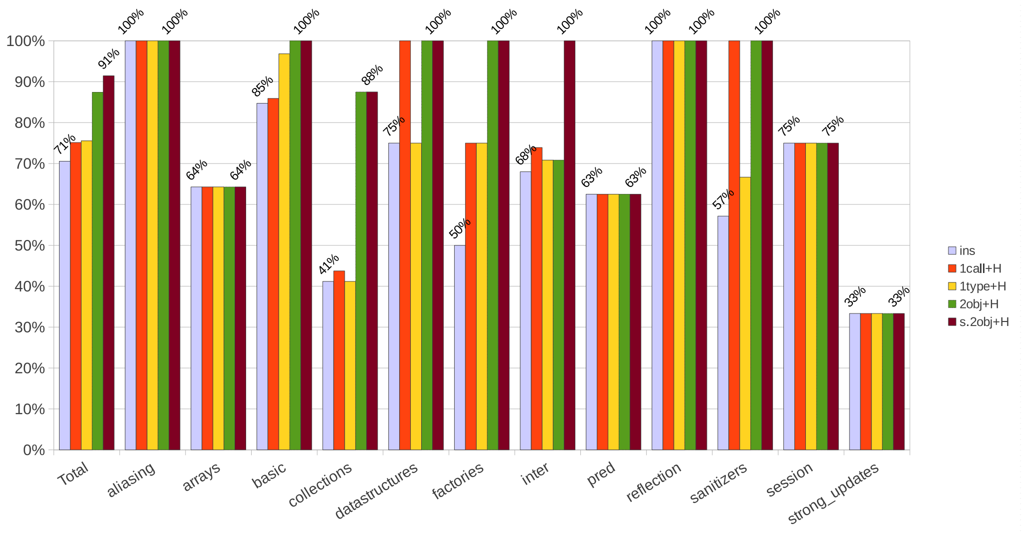


Fig. 8. Precision metrics for SecuriBench Micro under different parameters of context sensitivity (e.g., ins is a context insensitive analysis). Only leftmost and rightmost data labels shown for readability.

or "pred" suites), as well as 86% precision and 93% recall on an earlier version of DroidBench (containing 35 benchmarks out of the 118 that exist today).

Additionally, P/Taint is highly efficient (**RQ.D**). The analysis of Fig. 6 took just over 7 minutes to analyze 122 servlet benchmarks. DroidBench benchmarks were analyzed individually, with each taking under 150sec.

We next examine in more detail the precision, recall, and efficiency, when the analysis settings change. We focus on the SecuriBench Micro and JInfoFlow-bench suites because they can be run in a brief amount of time.

7.1.2 Sensitivity Analysis: Precision. Since context sensitivity is the main precision enabler in points-to analysis, we ran the SecuriBench Micro and JInfoFlow-bench benchmarks for different flavors of context sensitivity, as well as for a context-insensitive analysis. Fig. 8 shows the different precision scores for the SecuriBench Micro suite and Fig. 9 does the same for the rest of the benchmarks.

The figures clearly show the impact of context sensitivity on precision. A context-insensitive analysis would yield overall precision of 70% for SecuriBench and just above 80% for the rest of the benchmarks. With our unified analysis approach, employing higher precision comes at no development cost, and modifying the precision/scalability trade-off is a simple matter of picking a different setting of context sensitivity.

A few other interesting insights emerge:

- Unsurprisingly, as the level of context sensitivity is improved, from context-insensitive (ins) to selective hybrid 2-object sensitive + heap, we get better precision. However, some context sensitivities are better than others under different circumstances. For instance type sensitivity is better for than call-site sensitivity for SecuriBench, but not for the rest of the benchmarks.

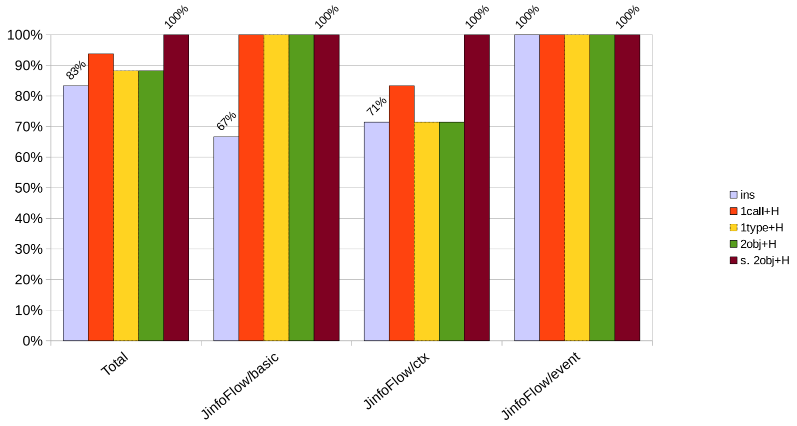


Fig. 9. Precision metrics for JInfoFlow-bench under different analysis context sensitivity.

- In P/Taint, context sensitivity helps us achieve greater precision even in benchmarks that were not originally designed to exercise context sensitivity. P/Taint does away with a number of predefined black box taint transfer functions designed to map information flow through complex object structures in the JRE. Instead, P/Taint relies on its full semantic support of the Java language and good context sensitivity.

7.1.3 Sensitivity Analysis: Recall. Reflection analysis is the main feature that enables higher recall in realistic information-flow benchmarks. We ran P/Taint with and without reflection support to compare recall. The analysis leverages Doop's advanced reflection support [Smaragdakis et al. 2015]: it performs substring analysis (considers which string constants in the program could be parts of a class, method, or field name in a reflection operation) and use-based analysis (considers where the results of reflection operations are used, to infer what they must have been).

In SecuriBench Micro, for the setup of our earlier results in Fig. 6, recall dropped from 99.3% to 97.2% when disabling reflection. Other flavors of context sensitivity experience near-identical differences in recall and are not shown. Generally, securibench only has one sub-suite affected by reflection so the overall numbers are not affected significantly.

The JInfoFlow-bench benchmarks have more interesting reflection-related behavior. We show the impact of reflection support on the overall recall of the information-flow analysis in Fig. 10. As can be seen, two of the sub-suites are significantly impacted by having static reflection support.

7.1.4 Sensitivity Analysis: Running Time. Fig. 11 shows the time required to analyze the various benchmark suites under different analysis parameters. Note that the two parts of the figure are not on the same scale: the support for open programs (such as SecuriBench's servlets) has made it possible to analyze all SecuriBench programs simultaneously, resulting in fast analysis times (Fig. 11a). For the other suites, programs were analyzed individually and the overall time reached several tens of minutes.

We can see clearly how the choice of context sensitivity influences analysis performance. Even more strikingly, however, reflection support roughly doubles the execution time of the analysis. Still, overall, the framework's performance is high, with running time scarcely being a concern.

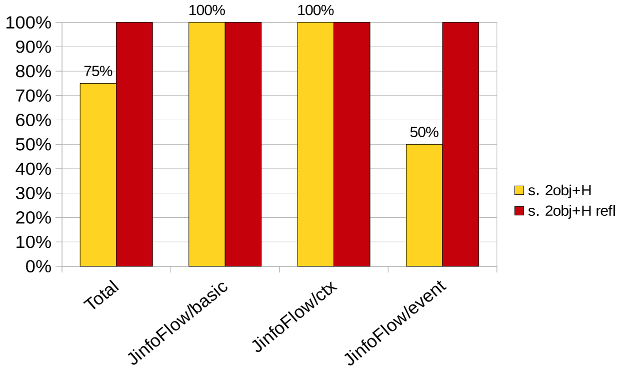
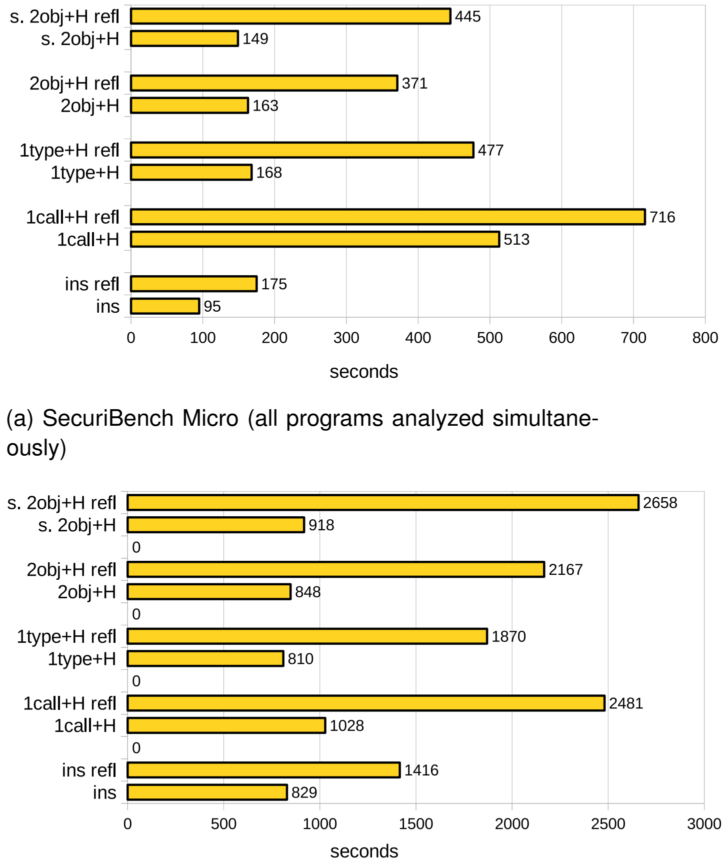


Fig. 10. Recall for JInfoFlow-bench with and without reflection.



(a) SecuriBench Micro (all programs analyzed simultaneously)

(b) JInfoFlow-bench (each program analyzed individually)

Fig. 11. Total run time for the analysis of benchmarks suites under different flavors of context sensitivity, with and without reflection analysis.

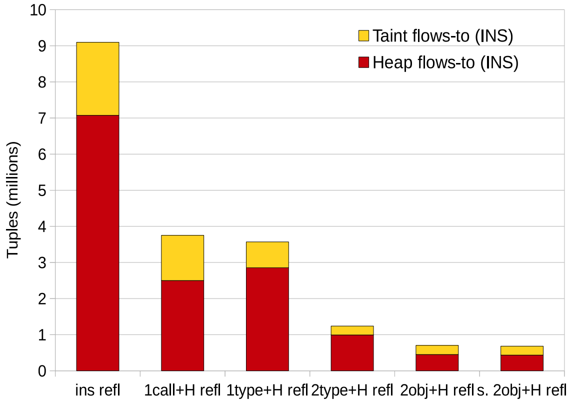


Fig. 12. Sizes of FLOWSTOVAR relation, split by heap objects and tainted objects.

7.1.5 Unified Analysis vs. Points-To. Finally, we report some metrics on the performance and internal complexity of the unified points-to/information-flow analysis, compared with the original underlying points-to analysis. Recall that the algorithmic logic for the core of the two analyses is identical, yet it now processes more abstract objects. It is interesting to see the relative proportion of "flows to" inferences that pertain to taint abstract objects vs. those that pertain to heap abstract objects.

Fig. 12 shows (for SecuriBench) the size of the FLOWSTOVAR relation (with context projected away, i.e., only the final useful information remaining), broken up into tuples that refer to abstract taint values and tuples that refer to abstract heap values. Across multiple analyses, of varying precision, we can see that the flow of taint objects is a significant part of the overall computation, up to about 36% for two object-sensitive analyses and 33% for the call-site sensitive analysis. This confirms that taint propagation is not a mere side-task for the underlying analysis logic—even with relatively few taint sources, it often rivals the inferences made regarding the flow of heap abstract objects.

7.1.6 Summary of Controlled Benchmarking. Overall, our experiments show P/Taint to perform very well, with over 91% precision and near-perfect (over 99%) recall for SecuriBench, 86% precision and 96% recall for DroidBench, and perfect scores for JInfoFlow-bench. The analysis clearly benefits from the precision and recall enhancements offered for free by a mature points-to analysis framework. Furthermore, the framework is efficient, scaling with ease to all the benchmarks.

7.2 Real-World Android Applications

A second part of our experimentation consists in evaluating P/Taint over a selection of large, popular Android applications: Google Chrome, Facebook Messenger, Google Translate, Instagram, Pinterest, Pokeradar, S Photo Editor, WhatsApp. These are apps in-the-wild, downloaded from Google Play. Additionally, they were analyzed with a full Android 25 SDK (i.e., with the library built from source code and fully analyzed, in conjunction with the application).

Owing to the good soundness and precision of P/Taint, we were able to find interesting program behavior and potential leaks. Figure 13 shows characteristics of the benchmarks examined, metrics on the internal analysis complexity (e.g., total flows-to), and analysis time. The "analyzed methods" are those that the analysis found the need to examine—these typically range from 25% to 50% of the total application methods.

App.	Version	Analyzed Methods	Total Flows-To	Tainted Flows-To	Time (s)
Chrome	57.0.2987.132	39,944	56,168,662	9,769,307	2,606
FB Messenger	108.0.0.20.70	127,843	886,331,909	42,146,495	22,924
Translate	5.8.0	38,446	58,269,015	7,871,702	1,846
Instagram	10.5.1	50,552	108,152,452	16,732,102	3,918
Pinterest	6.13.0	49,401	105,419,805	10,697,848	4,167
Pokeradar	1.4-4	33,002	33,163,210	1,671,242	1,004
S Photo Editor	1.07	45,440	42,710,031	4,047,319	1,068
WhatsApp	2.17.79	56,862	174,409,922	52,644,141	9,551

Fig. 13. Analysis metrics for popular Android applications.

Since we have no source-code access for these applications, it is nearly impossible to classify leaks beyond an intuitive, high-level understanding. However, we found even such an understanding to yield insights. For instance, most apps have been found to leak information such as location data to the network or file system.

In the Pokeradar Android client, there are 20 distinct instances where Google Analytics or other ad libraries are found to send location information over the web. Fortunately, no instances of personal information flowing to telephony or network APIs were recorded. The same patterns were found on other popular apps such as WhatsApp. Our analysis also spotted benign SMS message information (account verification codes) flowing to WhatsApp servers, and personal information flowing from Google drive (via broadcast receivers) to WhatsApp servers. A worrying flow that was uncovered was from Google drive data (also via broadcast receivers) that is used to determine the URL from which additional code is loaded at run time. Without code access it is not clear whether an attack vector might exist whereby a third party may tamper with the unencrypted[7] WhatsApp backups on Google drive to influence what code is loaded at run time. However, these are exactly the kinds of patterns that a software engineer would likely be prompted to examine more closely.

Perhaps most importantly, the Android applications test the scalability of P/Taint on realistic code bases, including the full Android library (not merely API stubs). Figure 13 shows analysis times that range from a few minutes to 7 hours, for the Facebook Messenger app, which contains over 55,000 classes. The analysis employed is a 1-call-site-sensitive analysis with partial context (only applied when calling methods that can return references to the objects that were passed in).

Overall, we see that P/Taint applies to programs of realistic scale and can leverage its big advantages of high configurability (e.g., of analysis context-sensitivity) by transparently inheriting its algorithmic logic from an underlying points-to framework.

8 RELATED WORK

There is substantial related work in both modern information-flow analysis techniques and under the theme of analysis-combination ideas. We next select some representative samples.

Livshits [2006] has also fruitfully explored the use of Datalog for taint analysis, but without elements of unifying the approach with pointer analysis.

Tripp et al. [2013] express taint analysis as a demand-driven problem, instead of a complete all-program-points flow. This is an interesting concept and certainly one worth exploring in future work. Their Andromeda tool has multiple language support, applying to Java, .NET, and JavaScript. Compared to our approach, the work offers no unification of analyses, with taint analysis requiring extra data-flow equations [Tripp et al. 2013, Fig.2,3] and separate concepts, such as taint sets. If one were to add context sensitivity, all rules would need to be enhanced to carry the same context

[7] https://www.whatsapp.com/faq/en/android/28000019

information as rules that determine the flow of abstract values. In our approach, only a single set of rules compute value flow, be it for regular object values or taint values. Thus, our information-flow analysis inherits context sensitivity support from the underlying points-to analysis without extra effort.

There are many Android taint analysis frameworks, and FlowDroid [Arzt et al. 2014] is one of the more successful tools. It can perform a flow-sensitive static taint analysis, specifically designed for Android applications. FlowDroid expresses its information-flow logic separately from an alias analysis and call-graph construction logic. The system models many features of Android, such as callbacks, and can model specific user interface elements as sources and sinks. Supporting features such as callbacks is not trivial and requires an incremental call-graph construction and addition of entry points at each iteration. FlowDroid relies on a manually-compiled list of more than 700 lines of sources, sinks, and taint transfer methods. Taint transfer methods include most operations on common collections. This may suggest a lack of generality in the core analysis. StubDroid [Arzt and Bodden 2016] partially addresses this problem by automatically compiling summaries of taint transfer methods. Naturally, positive effects on performance have been reported following the application of these summaries.

IccTA is an improvement on the original FlowDroid, that allows taint propagation between components [Li et al. 2015a]. Inter-component communication (ICC) in Android manifests itself as an information-flow discontinuity during analysis. This is addressed in IccTA by generating stub methods that connect components together and analyzing this modified program instead. Another Android analysis framework is DroidSafe [Gordon et al. 2015], which is similar in approach, except that it benefits from deep levels of object sensitivity, rather than flow sensitivity, in its core static analysis. The authors have also developed an Android run time for use during analysis that explicitly captures the semantics of life-cycle events.

Another approach to information-flow analysis is via the use of program dependence graphs, as in Pidgin [Johnson et al. 2015]. Program dependence graphs can be used to model any form of data and control dependence between instructions. By also detecting control dependencies, PDGs are a good choice if detection of implicit flow of information is important. In our earlier experiences while developing P/Taint we found that control-flow dependencies lead to higher levels of imprecision and hence we do not consider control-flow dependencies. Pidgin also supports multiple flavors of context sensitivity, which can also mitigate this imprecision. DroidInfer [Huang et al. 2015] performs taint analysis using constraint flow graph reachability algorithms. It relies on WALA[8] to produce a control-flow graph prior to performing its main analysis. Yet another approach involves the use of program slicing [Tripp et al. 2009], with an efficient representation for multiple program slices with a lot of common nodes. This approach is also a client of context-sensitive pointer analysis.

Much recent interesting work centers on dynamic languages. TAJS [Jensen et al. 2009] is one of the first and best-known JavaScript analysis frameworks, aiming to find type-related errors using static analysis. Dahse and Holz [2014] present an advanced technique for detecting second-order vulnerabilities (a payload is first stored, then accessed). Their work is in the context of PHP, as is the work of Hauzar and Kofron [2015] which combines value and heap analysis for detecting security vulnerabilities. The latter is an instance of a fruitful combination (though not unification) of static analyses.

A general analysis-combination pattern is that of performing two analyses in an intertwined fashion, so that the results of one can feed into the other. Instances of this pattern are *on-the-fly call-graph construction* (e.g., [Lhoták 2006]) and *on-the-fly exception analysis* [Bravenboer and Smaragdakis 2009a]. The unification of points-to and information-flow analysis is different in that it

[8] http://wala.sourceforge.net

is leveraging the logic of one analysis to perform the other, and not merely evaluating two separate analysis simultaneously. Furthermore, the underlying points-to analysis results are not affected by the addition of taint abstract objects, unlike in past on-the-fly analyses.

Volpano et al. [1996] offered a well-known formulation of the problem of secure information flow as a type system. This is an instance of expressing information-flow concepts in a known formal framework, rather than a combination of analyses. Notable differences from our work are that the type system does not treat pointers and there is no use of existing type system algorithms to also compute information flow.

More generally, abstracting away from a specific analysis is a time-honored pattern in static analysis research, and has yielded some of the deepest work in the area. Certainly, the *abstract interpretation* framework for specifying static analyses [Cousot and Cousot 1977, 1979] has strong elements of unifying analyses. More recently such elements have been brought out further, in the "abstracting abstract machines" work [Van Horn and Might 2010], which offers a general recipe for deriving static analyses from well-known abstract machines. However, our unification of points-to and information-flow analysis is quite different: it is not merely an effort to exploit commonalities in the two analyses specifications, but a complete integration into one analysis with essentially unchanged logic.

9 CONCLUSION

In this paper we proposed unifying points-to analysis and information-flow analysis, by leveraging algorithms of the former to implement the latter. The unification approach underlies the design of P/Taint: a framework that implements information-flow analysis by coercing a sophisticated pointer analysis framework (Doop) into computing additional data flow facts. We achieved this goal by elevating the domain of abstract heap objects and making small modifications to add aspects of taint analysis with no counterparts in pointer analysis (taint transfer and sanitization). The results have been quite impressive (precision, recall, run time), especially considering the extent of leveraging algorithms developed for different purposes (points-to analysis).

While program analysis designers have viewed information and points-to analysis as distinct processes, our work shows that the two can be unified fruitfully. The approach is not merely possible but compelling for several reasons: engineering effort, language semantics support, precision. In addition, we have shown (via modification of a highly abstract model of points-to analysis) that the approach is also general enough to be applied to multiple analysis algorithms. Practical applications of this approach to other pointer analysis frameworks may follow. As part of our future work we will be investigating how support for Android and more open programs can be added.

ACKNOWLEDGMENTS

We gratefully acknowledge funding by the European Research Council, grant 307334 (SPADE). In addition, the research work disclosed is partially funded by the REACH HIGH Scholars Program – Post-Doctoral Grants. The grant is part-financed by the European Union, Operational Program II, Cohesion Policy 2014-2020 (Investing in human capital to create more opportunities and promote the wellbeing of society - European Social Fund).

REFERENCES

Steven Arzt and Eric Bodden. 2016. StubDroid: automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 725–735. DOI: http://dx.doi.org/10.1145/2884781.2884816

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint

Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. DOI: http://dx.doi.org/10.1145/2594291.2594299

Martin Bravenboer and Yannis Smaragdakis. 2009a. Exception Analysis and Points-to Analysis: Better Together. In *Proc. of the 18th International Symp. on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 1–12. DOI: http://dx.doi.org/10.1145/1572272.1572274

Martin Bravenboer and Yannis Smaragdakis. 2009b. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '09)*. ACM, New York, NY, USA.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*. 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 269–282.

Johannes Dahse and Thorsten Holz. 2014. Static Detection of Second-order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 989–1003. http://dl.acm.org/citation.cfm?id=2671225.2671288

Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*.

David Hauzar and Jan Kofron. 2015. Framework for Static Analysis of PHP Applications. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 689–711. DOI: http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.689

Michael Hind. 2001. Pointer analysis: haven't we solved this problem yet?. In *Proc. of the 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*. ACM, New York, NY, USA, 54–61.

Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and Precise Taint Analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 106–117. DOI: http://dx.doi.org/10.1145/2771783.2771803

Neil Immerman. 1999. *Descriptive Complexity*. Springer. DOI: http://dx.doi.org/10.1007/978-1-4612-0539-5

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS) (LNCS)*, Vol. 5673. Springer-Verlag.

Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. 2015. Exploring and Enforcing Security Guarantees via Program Dependence Graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 291–302. DOI: http://dx.doi.org/10.1145/2737924.2737957

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. *Soufflé: On Synthesis of Program Analyzers*. Springer International Publishing, Cham, 422–430. DOI: http://dx.doi.org/10.1007/978-3-319-41540-6_23

Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-chieh Shan. 2012. An Analysis of the Mozilla Jetpack Extension Framework. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 333–355. DOI: http://dx.doi.org/10.1007/978-3-642-31057-7_16

George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-To Analysis. In *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA.

Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA.

Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. 2014. FlowTwist: Efficient Context-sensitive Inside-out Taint Analysis for Large Codebases. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 98–108. DOI: http://dx.doi.org/10.1145/2635868.2635878

Ondřej Lhoták. 2006. *Program Analysis using Binary Decision Diagrams*. Ph.D. Dissertation. McGill University.

Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015a. IccTA: Detecting Inter-component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 280–291. http://dl.acm.org/citation.cfm?id=2818754.2818791

Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-Inferencing Reflection Resolution for Java. In *Proc. of the 28th European Conf. on Object-Oriented Programming (ECOOP '14)*. Springer, 27–53.

Yue Li, Tian Tan, and Jingling Xue. 2015b. Effective Soundness-Guided Reflection Analysis. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings (Lecture Notes in Computer Science)*, Sandrine Blazy and Thomas Jensen (Eds.), Vol. 9291. Springer, 162–180. DOI: http://dx.doi.org/10.1007/

978-3-662-48288-9_10

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. *The Java Virtual Machine Specification, Java SE 8 Edition*.

Benjamin Livshits. 2006. *Improving Software Security with Precise Static and Runtime Analysis*. Ph.D. Dissertation. Stanford University.

Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection Analysis for Java. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*. Springer, 139–160. DOI:http://dx.doi.org/10.1007/11575467_11

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proc. of the 2002 International Symp. on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 1–11. DOI:http://dx.doi.org/10.1145/566172.566174

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (2005), 1–41. DOI:http://dx.doi.org/10.1145/1044834.1044835

Durica Nikolić and Fausto Spoto. 2012. Definite Expression Aliasing Analysis for Java Bytecode. In *Proc. of the 9th International Colloquium on Theoretical Aspects of Computing (ICTAC '12)*, Vol. 7521. Springer, 74–89. DOI:http://dx.doi.org/10.1007/978-3-642-32943-2_6

Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 111–124. DOI:http://dx.doi.org/10.1145/2676726.2677009

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 419–428. DOI:http://dx.doi.org/10.1145/2594291.2594321

Barbara G. Ryder. 2003. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In *Proc. of the 12th International Conf. on Compiler Construction (CC '03)*. Springer, 126–137. DOI:http://dx.doi.org/10.1007/3-540-36579-6_10

Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2011. Refactoring Java Programs for Flexible Locking. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 71–80. DOI:http://dx.doi.org/10.1145/1985793.1985804

Micha Sharir and Amir Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program flow analysis: theory and applications*, Steven S. Muchnick and Neil D. Jones (Eds.). Prentice-Hall, Inc., Englewood Cliffs, NJ, Chapter 7, 189–233.

Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. Dissertation. Carnegie Mellon University.

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and TrendsÂő in Programming Languages* 2, 1 (2015), 1–69. DOI:http://dx.doi.org/10.1561/2500000014

Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *Proc. of the Asian Symp. on Programming Languages and Systems (APLAS '15)*. Springer.

Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 17–30.

Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Alias Analysis for Object-Oriented Programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer Berlin Heidelberg, 196–232. DOI:http://dx.doi.org/10.1007/978-3-642-36946-9_8

Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*. Springer-Verlag, Berlin, Heidelberg, 210–225. DOI:http://dx.doi.org/10.1007/978-3-642-37057-1_15

Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 87–97. DOI:http://dx.doi.org/10.1145/1542476.1542486

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 51–62. DOI:http://dx.doi.org/10.1145/1863543.1863553

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2-3 (Jan. 1996), 167–187. http://dl.acm.org/citation.cfm?id=353629.353648

Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. 2012. Sound and Precise Analysis of Parallel Programs Through Schedule Specialization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 205–216. DOI:http://dx.doi.org/10.1145/2254064.

2254090