IFDS Taint Analysis with Access Paths

Nicholas Allen, François Gauthier, Alexander Jordan

March 18, 2021

Abstract

Over the years, static taint analysis emerged as the analysis of choice to detect some of the most common web application vulnerabilities, such as SQL injection (SQLi) and cross-site scripting (XSS) [OWA]. Furthermore, from an implementation perspective, the IFDS dataflow framework [RHS95] stood out as one of the most successful vehicles to implement static taint analysis for real-world Java applications [TPF+09, TPC+13, ARF+14].

While existing approaches scale reasonably to medium-size applications (e.g. up to one hour analysis time for less than 100K lines of code), our experience suggests that no existing solution can scale to very large industrial code bases (e.g. more than 1M lines of code). In this paper, we present our novel IFDS-based solution to perform fast and precise static taint analysis of very large industrial Java web applications.

Similar to state-of-the-art approaches to taint analysis, our IFDS-based taint analysis uses *access paths* to abstract objects and fields in a program. However, contrary to existing approaches, our analysis is demand-driven, which restricts the amount of code to be analyzed, and does not rely on a computationally expensive alias analysis, thereby significantly improving scalability.

1 Background

The IFDS analysis framework is a dataflow analysis framework for solving inter-procedural, finite, distributive, subset (IFDS) problems. Flow functions f are defined over a finite domain of dataflow facts D, and have to be distributive over the meet operator, union (i.e. $f(a) \cup f(b) = f(a \cup b)$). These flow functions are defined by the specific analysis (in our case, taint analysis), to specify the effect on dataflow facts that corresponds with the execution of the statement at the given program point. The IFDS analysis framework solves dataflow problems efficiently by reducing them to graph reachability problems. The reachability of a particular node in the graph represents whether a particular dataflow fact holds at a particular program point.

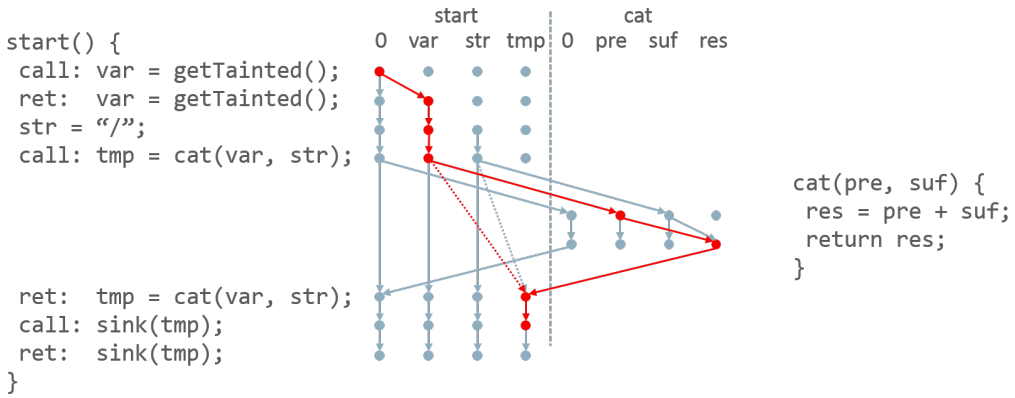

Figure 1: Simple IFDS Taint Analysis Example

There are two main variants of the IFDS analysis algorithm. The forward version of the analysis propagates facts forward through the program and exhaustively computes the dataflow facts that hold at each program point. In contrast, the backward version of the analysis is demand-driven, i.e., whether a particular fact holds at a particular program point is determined *on-demand*, in response to client queries. Our work is based on the backward version of the IFDS analysis algorithm.

The IFDS analysis framework achieves efficient inter-procedural analysis via function summarisation. Function summaries are generated on demand during the analysis, and represent the backward reachability from an end fact to a set of start facts. There can be multiple summaries generated for a single function, one for each relevant end fact.

IFDS operates on the *exploded supergraph* of a program, which is an inter-procedural control-flow graph (ICFG) where each node in the ICFG is exploded into as many nodes as there are dataflow facts. Figure 1 shows an example of an IFDS analysis (in this case, a simple taint analysis) applied to the following program:

```
1  void start() {
2    String var = getTainted();
3    String str = "/";
4    String tmp = cat(var, str);
5    sink(tmp);
6  }
7  String cat(String pre, String suf) {
8    String res = pre + suf;
9    return res;
10 }
```

A particular dataflow fact holds at a particular statement if and only if the associated node in the exploded supergraph is reachable. Edges in the exploded supergraph encode the flow

functions. In Figure 1, the fact that the sink node (`sink(tmp)`, *tmp*) is reachable from the entry node (`var = getTainted()`, **0**) indicates the existence of an execution path, highlighted in red, where tainted data reaches a security-sensitive sink.

In our approach, the exploded supergraph is extracted from programs in SSA form, and we use IFDS to encode a taint analysis over k-bounded access paths.

SSA Form We use *Static Single Assignment* (SSA) form [CFR+89] as an intermediate representation for our analysis. SSA requires that each use of a variable has a single definition. If there are multiple definitions for a use in the original program, a ϕ-function (or ϕ-node) is inserted in the control flow graph (CFG).

Access Paths Our analysis propagates *access paths* of the form $x.f.g$, where x is a local variable, and f and g are fields. Specifically, $x.f.g$ represents the value that is retrieved by first dereferencing x in the current scope and then dereferencing fields f and g from the heap. Because access paths are unbounded, we use *k-limiting* to bound their length to a pre-defined length k. When an access path reaches a length of k, further appends are simply ignored (i.e. access paths longer than k are assumed to be untainted). Our approach uses a default value of $k = 5$.

Taint Analysis The goal of taint analysis is to find and report dataflow from *sources* to security-sensitive *sinks* that does not undergo a *sanitisation* operation. In general, *taint labels* are used to track complementary taint information (e.g. SQLi vs. XSS). For the sake of simplicity, in this paper, we assume that values are either *tainted* or *untainted*, and that there are no sanitisation functions. Extending our approach to support sanitisation and taint labels is straightforward, and our implementation does support both of these features.

2 Approach

Our analysis performs a demand-driven, backward taint analysis. Similar to recent work on IFDS-based static taint analysis [TPC+13, ARF+14], our analysis tracks taint through objects and fields by propagating *access paths*.

In our implementation, we adapted the extended forward IFDS algorithm presented by Naeem et al. in [NLR10] to the on-demand backward analysis presented by Reps et al. in [RSH94]. Our extended algorithm adds the following three optimisations described in [NLR10] to the original backward IFDS algorithm:

1. Lazy computation on the exploded supergraph (e.g. the graph with one node per instruction and dataflow fact, as shown in Figure 1). Lazy computation of the exploded supergraph reduces the memory footprint of the analysis by ensuring that only the relevant portions of the exploded supergraph are built.

2. Support for ϕ-nodes in SSA form. Because ϕ-nodes at merge points in the CFG cause a loss of precision during dataflow analysis, this extension ensures that the IFDS algorithm delays merging of dataflows until *after* ϕ-nodes have been processed.

3. Providing the procedure-call flow function (in [NLR10] this was applied to procedure-return in the forward version of the analysis) with information about the caller-side state from the time of the procedure-return, allowing the callee-side state to be mapped to the caller-side context more precisely.

We now present the intra-procedural flow functions that define our backward IFDS taint analysis. Algorithm 1 defines (in the FLOW procedure) the flow functions for allocation, assignment, field-load and field-store statements in a Java program. For each type of statement ($[\![stmt.]\!]$) the flow function defines which facts must hold (any) *before* the statement, for a given fact to hold *after* the execution of the statement. And because our analysis operates on access paths, a flow function maps an access path of the form $b.f_1 \dots f_n$, where b is the base variable, and $f_1 \dots f_n$ is a sequence of fields, to a set of access paths. Note that we omit the inter-procedural call and return flow functions, because they simply convert arguments and return values between callers and callees without modifying access paths. The FLOW procedure is invoked during the execution of our backward IFDS algorithm implementation as statements are processed, to perform on-the-fly exploded supergraph construction for intra-procedural dataflow edges.

Case 1 defines the flow function for allocation statements. The incoming access path is mapped to the empty set ($\emptyset$) if its base variable b matches the newly assigned local variable x to capture the fact that access paths rooted at x cannot exist before x is allocated. Otherwise, the identity function is applied. Case 2 defines the flow function for assignments of the form $x = y$. The base variable b of the incoming access path is replaced with y if b matches x. Case 3 defines the flow function for assignment of tainted values. If b matches x, the incoming access path is mapped to the null fact ($\mathbf{0}$), to capture the fact that x became tainted at that specific point in the program.

Cases 4 and 5 define the flow functions for loads of the form $x = y.g$ and stores of the form $x.g = y$, respectively. Because our algorithm works with programs in the SSA intermediate representation (IR), care must be taken to reify statements involving multiple stores and loads. Indeed, translation to an IR usually deconstructs field accesses into multiple sub-statements using temporary variables that require reification before analysis. To address this issue, our analysis performs an on-demand, intra-procedural reification step (the REIFY procedure) before processing any store or load instruction, which determines the full access path referenced by the load or store statement.

Hence, Case 4 defines the flow function for loads of the post-reification form $x = z.g_1 \dots g_m$. The base variable b is replaced with z, and the loaded fields $g_1 \dots g_m$ are prepended to the access path if b matches x (unless the length of the new access path exceeds the pre-defined limit k, in which case the empty set is returned). Case 5 defines the flow function for stores of the post-reification form $z.g_1 \dots g_m = y$. The base variable b is replaced with y, and fields $f_1 \dots f_m$ are removed from the incoming access path if b matches z and the stored fields $g_1 \dots g_m$ match $f_1 \dots f_m$ (i.e. the stored fields form a prefix of the incoming access path). If

Algorithm 1 Intra-procedural flow functions

```
constant k

procedure FLOW(statement, (b.f_1 ... f_n))
    match statement
        case [[x = new]]                                                    ▷ (1)
            if x = b then return ∅
            else return {(b.f_1 ... f_n)}

        case [[x = y]]                                                      ▷ (2)
            if x = b then return {(y.f_1 ... f_n)}
            else return {(b.f_1 ... f_n)}

        case [[x = TaintSource()]]                                          ▷ (3)
            if x = b then return {0}
            else return {(b.f_1 ... f_n)}

        case [[x = y.g]]                                                    ▷ (4)
            if x = b then
                z.g_1 ... g_m ← REIFY((y.g))
                if m + n > k then return ∅
                else return {(z.g_1 ... g_m.f_1 ... f_n)}
            else return {(b.f_1 ... f_n)}

        case [[x.g = y]]                                                    ▷ (5)
            z.g_1 ... g_m ← REIFY((x.g))
            if z = b and m ≤ n and g_1 ... g_m = f_1 ... f_m then
                if ∀i ∈ [1, m], g_i is not an array then return {(y.f_{m+1} ... f_n)}
                else return {(y.f_{m+1} ... f_n),(b.f_1 ... f_n)}
            else return {(b.f_1 ... f_n)}


procedure REIFY((b.f_1 ... f_n))
    match DEFINITION(b)
        case [[b = y.g]]
            return REIFY((y.g.f_1 ... f_n))

        case default
            return (b.f_1 ... f_n)
```

any of the stored fields is an array, the incoming access is also preserved because our analysis is array-insensitive (e.g. it cannot reason about the exact array cell that is loaded), and hence cannot invalidate the incoming access path.

We now explain the reification step (the REIFY procedure) in more detail by way of an example. Assume that a is tainted, and that we are computing the flow function of the incoming access path $y.f.g.h$ and the statement `tmp2.h = a`. Without reification, Case 5 would wrongly conclude that `tmp2.h = a` has no impact on $y.f.g.h$ (as the base variables, $tmp2$ and y, do not match). To determine that store to $tmp2.h$ does, in fact, affect $y.f.g.h$, the reification step starts by tracking the definition of the base variable of the store/load. Then, if the definition is a load statement, the reification step replaces the base variable of the original store/load with the loaded access path, and starts tracking the definition of the base variable of the loaded access path. This is done recursively until it reaches a definition that is not a load statement. Once the reification step completes, the appropriate flow function can be applied to the reified store/load statement.

```
tmp1 = y.f;
tmp2 = tmp1.g
tmp2.h = a;
```

In our example, when processing the statement `tmp2.h = a`, the reification step would start by tracking the definition of the base variable $tmp2$. Then, it would replace $tmp2.h$ with $tmp1.g.h$, and start tracking the definition of $tmp1$. Finally, it would replace $tmp1.g.h$ with $y.f.g.h$. Thus, when the flow function defined in Case 5 is applied to the reified statement `y.f.g.h = a`, it correctly propagates taintedness from a to $y.f.g.h$.

2.1 Working Example

Figure 2 demonstrates our approach applied to an example program. In this example, the foo method obtains tainted data from the $getTainted$ method (the taint source), stores it into the field of a Box object ($box1$), then makes a copy of that object ($box2$), retrieves the data stored in the field of the copy and passes it to the $sink$ method (the taint sink).

To determine whether the data passed in at the sink, i.e. the $boxData$ variable, is tainted, the analysis works backward from the sink (line 32), tracking the dataflow fact $boxData$. In the prior statement (line 31), $boxData$ is assigned the return value of Box.get, so the summary for Box.get for the return value must be computed. The summarisation of Box.get starts at the return statement (line 10), tracking the dataflow fact str (the returned variable). In the prior statement (line 9), Case 4 applies, which maps back to the fact $this.f$, after which the method entry has been reached, so the summary generated for Box.get establishes flow to the return value from $this.f$. Returning to the call to Box.get (line 31), the summary is applied (substituting the actual $this$ object passed in), resulting in a transfer to the dataflow fact $box.f$. In the prior statement (line 29), $box2$ is assigned the return value of $copy$, so the summary for that method for the field f of the return value must be generated.

This summarisation of the $copy$ method starts at the return (line 20) with the dataflow fact $cpy.f$. In the prior statement (line 19), the $Box.put$ method is invoked on cpy, so the effect of method $Box.put$ on $this.f$ must be summarised. Case 5 is applied for the store statement in $Box.put$ and this summary produced is that $this.f$ flows from the argument str.

```
1  public class Box {
2    private String f;
3
4    public void put(String str) {     // 14. summary(this.f) = {arg0}
5      this.f = str;                   // 13. str
6    }                                 // 12. this.f
7
8    public String get() {             // 6. summary(<ret>) = {this.f}
9      String str = this.f;            // 5. this.f
10     return str;                     // 4. str
11   }                                 // 3. <ret>
12 }
13
14 public static Box copy(Box box) {   // 19. summary(<ret>.f) = {arg0.f}
15   Box cpy = new Box();              // 18. box.f
16                                     // 17. box.f
17   String data = box.get();          // 16. reuse summary(Box.get, <ret>)
18                                     // 15. data
19   cpy.put(data);                    // 11. compute summary(Box.put, this.f)
20   return cpy;                       // 10. cpy.f
21 }                                   // 9. <ret>.f
22
23 public static void foo() {
24   String tainted = getTainted();    // 24. 0 (null fact)
25   Box box1 = new Box();             // 23. tainted
26                                     // 22. tainted
27   box1.put(tainted);                // 21. reuse summary(Box.put, this.f)
28                                     // 20. box1.f
29   Box box2 = copy(box1);            // 8. compute summary(copy, <ret>.f)
30                                     // 7. box2.f
31   String boxData = box2.get();      // 2. compute summary(Box.get, <ret>)
32   sink(boxData);                    // 1. boxData
33 }
```

Figure 2: A simple program annotated with dataflow facts, as propagated by our algorithm. Numbers in the comment show the order in which statements are processed.

Returning to *copy* (line 19), the summary is applied, transferring to the fact *data* that is assigned the return of $Box.get$ in the prior statement (line 17). The already-computed summary for $Box.get$ is applied, transferring to $box.f$, which is unaffected by the prior new statement, so the summary generated for *copy* is that the field f of the return value flows from the field f of the argument.

Returning to foo (line 29), the summary for *copy* is applied, transferring from $box2.f$ to $box1.f$. At the prior statement (line 27), $Box.put$ is invoked on $box1$. The already computed summary for $Box.put$ is applied, transferring to *tainted*, which is unchanged until it is assigned the return value of $getTainted$ (line 24). For this example, $getTainted$ is designated as a taint source, so the dataflow fact *tainted* maps to the null fact.

The null fact always holds, and the analysis has demonstrated that the *boxData* fact at the sink is reachable from the null fact. Therefore, the *boxData* variable passed to the sink is tainted, and so a bug would be reported for this example.

3 Initial Results

Benchmark	Legacy Taint Analysis					IFDS-AP Taint Analysis				
	TP	TN	FP	FN	Runtime	TP	TN	FP	FN	Runtime
Securibench	99	0	10	39	0.1568	101	0	12	37	0.0184
WebGoat	35	0	1	33	0.5912	35	0	3	33	0.504
OWASP	501	533	451	324	3.7784	732	353	772	100	2.732

Table 1: Results for analysis benchmarks

Taint Analysis	TP	FP	Unknown	Runtime
Legacy	96	9	9	6m15s
IFDS-AP	121	14	41	3m3s

Table 2: Results for Oracle product A

Our technique has been implemented in the context of Parfait [CKL$^+$12], and applied to the task of detecting security vulnerabilities in Java EE web applications, such as SQL injections, cross-site-scripting, etc.

Table 1 shows the results of our analysis applied to three analysis benchmarks, Securibench, WebGoat and OWASP, compared with the results of our previously used taint analysis. Table 2 shows the comparison with the analysis applied to an Oracle product. The results demonstrate that our IFDS analysis detects more real bugs, and does so with significantly reduced runtime.

4 Related Work

Static taint analysis for the detection of vulnerabilities has a long history in the research community. In this section, we describe and contrast the state-of-the-art approaches that are most closely related to ours.

In [LL05], authors propose to use a flow-insensitive points-to analysis to support a client taint analysis for Java Enterprise Edition (JEE) web applications. While more modern approaches gained in scalability and precision, this paper was seminal and triggered a lot of follow-up research, as presented below.

TAJ [TPF+09] used an approach called *thin slicing* to perform taint analysis of JEE web applications. In thin slicing, IFDS is used for flow-sensitive reasoning about tainted flows through local variables while flows through the heap are handled by using flow-insensitive pre-computed points-to information. TAJ propagates taint information in a forward manner, from sources to sinks. Furthermore, TAJ bounds its analysis using various heuristics to keep its runtime and memory usage to acceptable levels.

Andromeda [TPC+13] first introduced the idea of using access paths in an IFDS-based setup to compute alias and taint analysis of JEE web applications simultaneously. In Andromeda, tainted access paths are propagated in a forward manner, from entry points of the program to security-sensitive sinks. Moreover, Andromeda also computes on-demand aliasing by launching a backward alias analysis whenever the forward analysis reaches an assignment to a field. Then, the forward taint analysis propagates taint through the newly discovered aliases, and so on until a fixed point is reached. Because the length of access paths is unbounded, Andromeda limits their length to a user-specified value k, a process known as *k-limiting*.

FlowDroid [ARF+14] integrated the dataflow equations of Andromeda into the IFDS framework and improved precision by sharing information between the taint and alias analyses. Indeed, in FlowDroid, aliases become tainted only *after* the original access path becomes tainted, a mechanism referred to as "activation statements" in the paper. Furthermore, FlowDroid includes support for Android-specific framework constructs that are hard to analyse statically. FlowDroid also uses a forward taint analysis combined with an on-demand backward alias analysis that uses *k-limiting*.

Boomerang [SDAB16] generalised the IFDS-based alias analysis of FlowDroid and decoupled it from the taint analysis. In Boomerang, client analyses can issue alias queries that will be solved on-demand. Boomerang also extends previous work by replacing access paths with access graphs that can represent multiple access paths of indefinite length. Otherwise, Boomerang reuses the forward and backward dataflow equations introduced in Andromeda [TPC+13] to compute alias information. When using Boomerang instead of its original alias analysis, FlowDroid could analyze more applications in a given timeout.

5 Comparison with Existing Approaches

An important component of our analysis is the ability to report bug traces for the analysis results. Indeed, several static program analyses do not keep track of *provenance* information and hence cannot produce explanations in the form of a bug trace from a sink to a source. Because IFDS is fully flow- and context-sensitive and because it stores provenance information in the form of *Path Edges*, our approach naturally produces understandable bug traces out-of-the-box.

Furthermore, contrary to existing approaches that are geared towards soundness, our taint analysis implementation is geared towards high scalability. For example, in our implementation, we use the *flyweight* [GHJV95] design pattern to ensure that each access path is created only once in memory and reused as many times as needed. Furthermore, we also optimise away nodes in the exploded supergraph that have only one predecessor and for which the transfer function is the identity function. Because most nodes fall in this category (e.g. most statements have only one predecessor and don't modify tainted access paths), this optimisation speeds up the analysis significantly. (We observed a speedup of up to 45% on large programs).

Moreover, contrary to [TPC+13, ARF+14] our taint analysis deliberately omits computing complete aliasing information, which would require an interplay between our backward taint analysis and a forward alias propagation analysis. This deliberate trade-off of soundness for scalability drastically reduces the theoretical complexity of our algorithm. Precisely, according to the definitions in [RHS95], our analysis is h-sparse, because, as shown in section 2, every transfer flow function produces at most 2 facts, and $2 << |D|$, where $|D|$ is the cardinality of the dataflow domain (e.g. all possible access paths in a program). According to [RHS95], *h-sparse* problems have a complexity of $O(Call\ D^3 + hED^2)$, where $Call$ is the number of call sites, D is the dataflow domain, and E is the set of intra-procedural edges. On the other hand, because the number of aliases of a given variable cannot be bounded to $h << |D|$ for non-trivial programs, an analysis that computes taint analysis together with an alias analysis is said to be *Distributive* and has a complexity of $O(ED^3)$.

While our technique and [TPC+13] both limit access paths to a maximum size of k, the approach used in [TPC+13] favours soundness by appending a Kleene star to access paths exceeding k, that are considered to match all other access paths sharing the same k-prefix. This may result in spurious taint flows being explored. In contrast, our *k-limiting* approach favours precision (and hence scalability, as fewer potential taint flows are explored), by ignoring any taint flows involving access paths exceeding k.

Table 3 summarizes the novelty of our approach with respect to state-of-the-art approaches.

References

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid:

Approach	Full alias analysis	Complete bug trace	Scalability	Complexity
TAJ	✓	✗	High (Bounded)	$O(ED^3)$
Andromeda	✓	✓	Medium	$O(ED^3)$
FlowDroid	✓	✓	Low	$O(ED^3)$
Our technique	✗	✓	Very high	$O(CallD^3 + 2ED^2)$

Table 3: Novelty of our approach

Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM Sigplan Notices*, 49(6):259–269, 2014.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. *Proceedings of the 16th Symposium on Principles of Programming Languages - POPL '89*, pages 25–35, 1989.

[CKL+12] Cristina Cifuentes, Nathan Keynes, Lian Li, Nathan Hawes, and Manuel Valdiviezo. Transitioning parfait into a development tool. *IEEE Security & Privacy*, 10(3):16–23, 2012.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[LL05] V Benjamin Livshits and Monica S Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[NLR10] Nomair A Naeem, Ondrej Lhoták, and Jonathan Rodriguez. Practical Extensions to the IFDS Algorithm. *CC*, 10:124–144, 2010.

[OWA] OWASP Top 10 Project. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`. Accessed: 19-12-2017.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1995.

[RSH94] Thomas Reps, Mooly Sagiv, and Susan Horwitz. Interprocedural dataflow analysis via graph reachability. Technical report, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark, 1994.

[SDAB16] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 22:1–22:26, 2016.

[TPC+13] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *16th International Conference on Fundamental Approaches to Software Engineering, FASE 2013*, 2013.

[TPF+09] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *ACM Sigplan Notices*, volume 44, pages 87–97. ACM, 2009.