FlowTwist: Efficient Context-Sensitive Inside-Out Taint Analysis for Large Codebases

Johannes Lerch[1], Ben Hermann[1], Eric Bodden[1,2] and Mira Mezini[1]
[1] Technische Universität Darmstadt, [2] Fraunhofer SIT
Darmstadt, Germany
{lastname}@cs.tu-darmstadt.de

ABSTRACT

Over the past years, widely used platforms such as the Java Class Library have been under constant attack through vulnerabilities that involve a combination of two taint-analysis problems: an integrity problem allowing attackers to trigger sensitive operations within the platform, and a confidentiality problem allowing the attacker to retrieve sensitive information or pointers from the results of those operations. While existing static taint analyses are good at solving either of those problems, we show that they scale prohibitively badly when being applied to situations that require the exploitation of both an integrity and confidentiality problem in combination. The main problem is the huge attack surface of libraries such as the Java Class Library, which exposes thousands of methods potentially controllable by an attacker.

In this work we thus present FlowTwist, a novel taint-analysis approach that works inside-out, i.e., tracks data flows from potentially vulnerable calls to the outer level of the API which the attacker might control. This inside-out analysis requires a careful, context-sensitive coordination of both a backward and a forward taint analysis. In this work, we expose a design of the analysis approach based on the IFDS algorithm, and explain several extensions to IFDS that enable not only this coordination but also a helpful reporting of error situations to security analysts.

Experiments with the Java Class Library show that, while a simple forward taint-analysis approach does not scale even with much machine power, FlowTwist's algorithm is able to fully analyze the library within 10 minutes.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Design, Languages, Performance, Security

Keywords

Taint analysis, confused deputy, IFDS

1. INTRODUCTION

Static taint analyses constitute an important class of static program analyses that track data to check whether it can flow from given sources to given sinks. Taint analyses are used to address integrity or confidentiality problems. When using the analysis for an integrity problem, sources are inputs that are controllable by an attacker and sinks are functions that perform sensitive operations. For confidentiality problems, sources are functions that may leak private data and sinks are outputs that an attacker can read.

Common vulnerabilities in systems that execute untrusted code often involve a combination of related integrity and confidentiality issues. Assuming these vulnerabilities have been introduced to a system's code accidentially, they are called *Confused Deputy Problems*[11]. In this class of problems an attacker uses a proxy possessing the necessary authority – the confused deputy – to carry out operations on its behalf. Detecting instances of this problem requires a combination of an integrity and a confidentiality analysis. Instances of the confused-deputy problem are a real and relevant problem for example in the Java Class Library (JCL). Vulnerabilities based on this problem have been exploited multiple times in the past. The most prominent exploits are described in the Common Vulnerabilities and Exposures Directory under the identifiers 2012-4681, 2013-0422, and 2013-2460 and have posed grave risk to the security of the Java platform. In the most common attack vector against Java, the attacker exploits the vulnerability to have the platform load a class on his behalf, which otherwise he would have no permission to load. In this kind of attack, the attacker controls the input to a class-loading method call, an integrity problem, and then retrieves the class object returned from that call, a confidentiality problem.

Existing taint-analysis approaches show often prohibitive scalability problems when considered as a means to discover the kind of problems described above, as we elaborate in the following.

A taint analysis for discovering integrity problems typically works in a forward mode: It starts at sources and follows assignments until finding a sink. If there are more sources than sinks, such a forward taint analysis may follow unnecessarily many paths which will never lead to a sink. These excess computations will eventually impede the scalability of such an analysis. In a real-world system that executes untrusted code, sources are all public API functions that the untrusted code can call. The amount of these public functions is typically by orders of magnitude larger than the number of sensitive sinks to control. For instance, in the

Java Class Library there are over 45,000 public methods. When considering an analysis to find vulnerabilities of calls to `Class.forName`, 134 call sites have to be regarded as sinks. Seeing all of the public methods of the JCL as sources, there are clearly more sources than sinks in an analysis of this kind.

A backward analysis starting at sinks can address the scalability issues of a forward taint analysis [12, 4]. However, this only provides a scalable solution to the integrity problem. For the confidentiality part of a confused-deputy problem, the backward analysis would now face the same problems described above: in principle, many methods of the public API would have to be considered as possible sources for the backward analysis, most of which would never probably reach a sink.

In this work we thus present FlowTwist, a novel analysis approach which efficiently solves the integrity part of a confused-deputy problem in a backward manner, and solves the confidentiality part just as efficiently in a forward manner. If applied to the Java Class Library, this analysis needs to start only at 134 potentially vulnerable `Class.forName` calls, and not at all 45,000 public API methods. The challenge for FlowTwist is to efficiently combine the analysis results of both analyses in a context-sensitive fashion such as to avoid spurious results caused by unrealizable control-flow paths. To reconstruct the context-sensitivity between the two independent analyses we match and connect a reconstruction of the call stacks of their results. Thereby, complete paths from sources to sinks and back towards the source are constructed, which are affected by an integrity and a confidentiality problem. We call this analysis an *inside-out analysis* as it starts at the inner layer of the API – which we consider as *sinks* – , e.g., a call to `Class.forName` within the Java runtime library, and then works its way outward to the public API – the *sources*.

We implement our approach by several extensions of the Interprocedural Finite Distributive Subset (IFDS) algorithm by Reps et. al. [19]. The original IFDS algorithm was designed to accommodate, for instance, forward taint analyses flowing from sources to sinks. Previous work [1] added the possibility to use the algorithm for backward analysis. In this work, we extend the IFDS algorithm with support for so-called unbalanced return flows, i.e., a flow out of a function whose calls have not been considered previously. Support for unbalanced return flows is necessary to facilitate an analysis that progresses from the inner layer of a API to the outer layer. Third, we extend the facts propagated by the algorithm to store information on the paths they were propagated on, to be able to reconstruct these paths.

This information is not only useful for reporting vulnerabilities, but it is also used by another novel algorithm which combines the results of the forward and backward taint analyses to derive a complete result containing code paths that are possibly vulnerable due to the combined integrity and confidentiality problem. The algorithm first constructs semi-paths from the two analyses results and then matches them to establish context-sensitivity. Matched semi-paths are combined to complete paths providing detailed information about the leaking data flows.

However, enumerating all semi-paths for matching threatens the scalability of the whole approach. Therefore, we introduce an additional extension to the IFDS algorithm that keeps the context of the forward and backwards analysis constantly in sync. As a result, the analysis algorithm will never produce semi-paths for the integrity problem for which no matching path of the confidentiality problem exists in the first place and vice versa, greatly reducing the number of paths that the path-matching algorithm needs to consider.

To validate our hypothesis that the inside-out approach is faster and scales better for large codebases than a pure forward taint analysis, we apply both analyses to the confused-deputy problem. All analyses implemented for the experiments are equally precise, context sensitive, flow sensitive, and report the same data flows. In our experiments we run them on the Java Class Library (JCL) shipped with the Java Runtime Environment. We consider the JCL as a large codebase as it contains over 18.500 classes with over 45.000 attacker-callable API methods.[1]

We find that our inside-out approach scales significantly better for an analysis of vulnerabilities to calls to `Class.forName` than a pure forward analysis, which we extended to cover integrity and confidentiality problems. In our experiments the inside-out approach was on average five times faster than a pure forward analysis, whilst using notably less memory. Also, we observe that only our dependent inside-out analysis is able to terminate in reasonable time for a larger problem targeted at finding vulnerabilities to calls to methods marked with the annotation `@CallerSensitive`. We therefore conclude that this approach effectively solves larger problems and thus scales better.
To recap, the main contributions of this paper are:

- A context-sensitive inside-out taint-analysis approach that solves the scalability problems of current approaches when applied to detect problems involving combined integrity and confidentiality issues in large codebases.
- An implementation of the proposed approach as an extension to the IFDS algorithm
- A novel path construction algorithm for more comprehensive reporting for the IFDS algorithm
- An extension to our analysis that effectively solves scalability problems in path construction
- An extensive evaluation of the proposed approach on a real problem scenario

The remainder of this paper is structured as follows. In Section 2 we give a brief overview of our approach. Section 2.1 presents and overview on the IFDS algorithm. In Section 3 we present the inside-out analysis approach that we propose in further detail. Section 4 presents the design, execution and the results of our evaluation. In Section 5, we discuss the relationship of other work. We conclude the paper in Section 6 with a summary of the findings and an outlook on possible future work.

2. OVERVIEW

In this section, we give background information on the IFDS algorithm [19] followed by a high-level overview of our proposal.

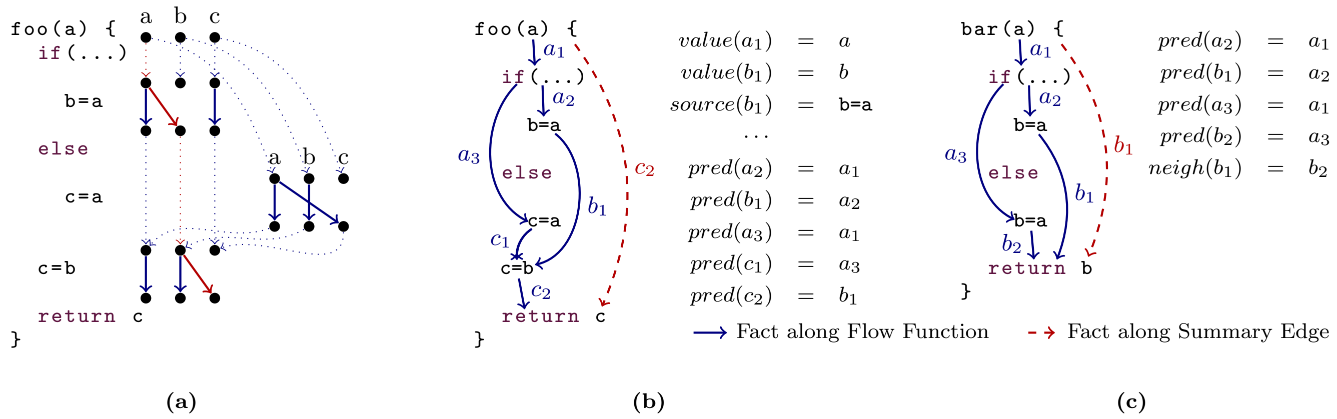


Figure 1: Fact Propagation in the IFDS Algorithm (**the relations** *predecessor* **and** *neighbors* **are shortened here as** *pred* **and** *neigh*)

```
    procedure PropagateAndMerge($\langle s_p, d_1\rangle \to \langle n, d_2\rangle$)
1:    if $\exists d_2' \mid$ value($d_2$) = value($d_2'$) $\wedge$
              $\langle s_p, d_1\rangle \to \langle n, d_2'\rangle \in$ Seen then
2:      $neighbors(d_2') := neighbors(d_2') \cup d_2$
3:    else
4:      Insert $\langle s_p, d_1\rangle \to \langle n, d_2\rangle$ into Seen
5:      Propagate($\langle s_p, d_1\rangle \to \langle n, d_2\rangle$)
6:    end if
    end procedure
```

Figure 2: Change to Support Neighbors in IFDS

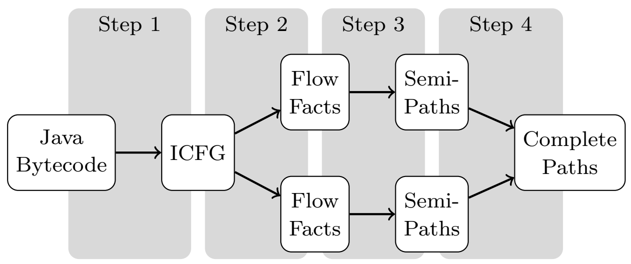


Figure 3: Overview of Steps Performed

2.1 IFDS Algorithm

The Interprocedural Finite Distributive Subset (IFDS) algorithm [19] addresses data-flow problems with distributive flow functions over finite domains. Reps et al. show that if the problem is modelled in this fashion, the analysis problem can be reduced to a graph reachability problem. The graph they built is a so called exploded super graph, in which for each node (s, d) is reachable from a special distinct start node if a data-flow fact d holds at a statement s.

For illustration, consider a taint analysis which should be applied to the example in Figure 1a. For the assignment `b = a` the flow function of a taint analysis should generate a fact that variable `b` is tainted after executing the statement, assuming variable `a` was tainted before. It also should retain the fact that variable `a` will then still be tainted. But, if variable `a` was not tainted before but variable `b` was, then variable `b` should not be tainted afterwards. Such semantics are typically illustrated by graphs as in the Figure 1a. Each dot represents a fact before and after the execution of a statement. Arrows between these dots represent that the applied flow function claims there is a data-flow from one fact to the other for a specific statement.

In the presented example there is a path shown in red color from a value passed as a parameter to the returned value. After analyzing the function `foo` once, the IFDS algorithm will create and store this path as a summary edge. This is important for scalability as it allows to reuse the summary edge the next time a call to `foo` is being processed instead of analyzing the whole function body again. This becomes even more obvious, when considering that `foo` could itself call other functions which in turn had to be analyzed again too.

2.2 The Proposal in a Nutshell

The suggested analysis performs four steps as illustrated in Figure 3. In the first step the static analysis framework Soot [14] is used to read Java Bytecode and to transform it to an intermediate three-address representation called Jimple. In that step it also computes an Interprocedural Control Flow Graph (ICFG). In the next step, the Interprocedural Finite Distributive Subset (IFDS) algorithm [19] uses the ICFG to compute data-flow facts along edges of the ICFG. Heros [1] is used as implementation of the IFDS algorithm, which is actually an implementation of the extended IFDS algorithm [17]. The IFDS algorithm is performed two times: one time for the integrity problem; and another time for the confidentiality problem. The IFDS algorithm implementation is extended and the data-flow facts are modelled such that they support reconstruction of exact paths along which the data-flow facts were propagated. These paths are reconstructed in the third step. However, we call these paths *semi-paths* as they represent only the integrity or confidentiality part of a complete path. In the last step, semi-paths are matched to generate complete paths.

3. ANALYSIS DESIGN

The analysis design we propose is capable of addressing integrity and confidentiality issues at the same time. Based on the observation that a pure forward analysis is not scalable or fast enough to be applied for such taint analysis problems

[1]Measured on Oracle Java 7 Update 25

on huge codebases, we propose an inside-out analysis design. Additionally, we address the lack of reporting along which statements a data-flow exists in the IFDS algorithm.

In Section 3.1 we first introduce how we model facts propagated by the IFDS algorithm to include information about statements and paths taken in data flows. An inside-out analysis requires the support of unbalanced returns, i.e., return flows for which no previous call flow has been registered. The original IFDS algorithm does not support such unbalanced flow. We therefore formalize them as an extension to IFDS in Section 3.2. After having executed the IFDS algorithm, FlowTwist constructs two classes of semi-paths, one for the integrity and one for the confidentiality problem. These are then matched pairwise to construct combined execution paths. Section 3.3 explains this path matching, followed by a discussion of simplifications to improve scalability in Section 3.4. Section 3.5 presents changes to the IFDS algorithm that make the analyses for integrity and confidentiality dependent on each other, avoiding analyzing and creating paths for program parts for which only an integrity *or* confidentiality issue exists. From a security point of view, such paths are not interesting, for instance because an attacker, while being able to trigger the loading of a restricted class, would not be able to obtain access to the loaded class handle.

We applied all extensions and changes to the IFDS algorithm implementation Heros [1] and provided these as contributions.[2] Additionally, we provide the implementation of our analysis.[3]

3.1 IFDS Extension to Store Path Information

The use of summary edges in IFDS has drawbacks when it comes to reporting analysis results. In the example in Figure 1a only one branch propagates a potential taint from the parameter to the returned value. Yet, information on possible alternative flows (or in this case their nonexistence) is lost. But not only information about these intraprocedural paths are lost, one also loses information about the interprocedural edges the analysis takes, as the summaries abstract over called procedures. Moreover, an IFDS-based analysis is only able to report the source and sink of a data flow, but not any intermediate statements.

To overcome this limitation, FlowTwist adopts a model of facts that allows to track the flow along which they are propagated. A natural approach to taint analysis with the IFDS algorithm is to use the identifiers of variables as propagated taint facts. However, to enable path tracking we need a more extensive fact representation. We propagate facts of type `Fact` instead and define several relations on this type:

$$\begin{aligned} value &: Fact \rightarrow Variable \\ source &: Fact \rightarrow Statement \\ predecessor &: Fact \rightarrow Fact \\ neighbors &: Fact \rightarrow \mathcal{P}(Fact) \end{aligned}$$

The relation *value* maps each fact to the related tainted variable. The relation *source* maps a fact to a statement at which the fact was generated. The relation *predecessor* links to the fact from which a flow function generated the current fact. Effectively, this creates a chain of facts, allowing to traverse the complete flow for a fact reported at an arbitrary sink. The relation *neighbors* links to similar facts, i.e., facts with the same *value*, at positions where flows are merged. Following examples will illustrate why this is model is simpler than storing multiple predecessors.

Figure 1b shows the propagated facts when modeled as described. Note that some propagated facts are left out to simplify the illustration. Consider how the fact that $predecessor(c_2)$ is b_1 and not c_1 encodes that the flow is only possible along one branch in `foo` (c.f. Figure 1a). The chaining of facts does not preclude the algorithm from computing summary functions, which is important for the scalability of the IFDS algorithm. In the example, the summary edge represents that if a tainted variable is passed as argument to `foo`, then the fact c_2 holds, i.e., `c` is tainted, when the method returns. The summary abstracts of the intermediate facts a_1, a_2, and b_1, but nevertheless the chain of predecessor links allows FlowTwist to later reconstruct the path along these facts through the reference to c_2, which is included in the summary.

To illustrate the role of the neighbors relation, consider the example in Figure 1c. Here, the facts b_1 and b_2 both hold at the same statement and both represent the fact that variable `b` is tainted. Moreover, these facts should be merged into a single one as otherwise from this point on, every propagation is computed two times for similar facts. This effect multiplies further for every branch taken, yielding a clear threat to the scalability of the analysis. IFDS is restricted to set union as a merge operator, and is thus unable to identify “similarity” of FlowTwist’s data-flow facts. Therefore, we extend the IFDS algorithm to recognize if a fact is propagated along an edge for which previously a fact was propagated with the same *value*. If this occurs, the second propagated fact is set to be a neighbor of the first propagated fact and the second fact is not propagated further. In contrast to creating predecessor links it is not possible to encode this behavior in a flow function. However, the IFDS algorithm can be extended by simply wrapping calls to the “Propagate” procedure [19] as shown in Figure 2. Given the fact d_2 to be propagated, “PropagateAndMerge” checks whether there is a fact d'_2 stored in the set *Seen* for which the *value* is the same as the *value* of d_2. If such a fact d'_2 exists, then d_2 is added to its neighbors and not propagated further; otherwise d_2 is added to the set *Seen* and propagated.

We do not use the *predecessor* relation to store information about a merge, as it increases complexity of handling summary edges. In the example function `bar` we would have to create two summary edges: one for the fact b_1 and one for fact b_2. For each caller of the function these have to be recognized as representing the same value, i.e., adding both as predecessors. Using the *neighbors* relation allows to store only one summary edge, i.e., for the first fact propagated to the return statement. Nevertheless, the path through the second fact can be equally reconstructed as it is stored as a neighbor of the first.

3.2 Unbalanced Return Flows

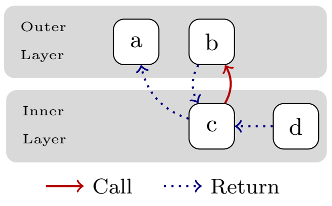
Neither in its original version [19] nor in its extended version [17] does the IFDS algorithm support unbalanced return flows. Unbalanced return flow occur when processing a return of a method for which no matching previous call was processed. In a typical context-sensitive analysis starting at the outer layer of an API, calls are always processed before

[2]The implementation of Heros is available on GitHub: `https://github.com/Sable/heros`

[3]The implementation of FlowTwist is available on GitHub: `https://github.com/johanneslerch/FlowTwist`

```
      procedure ForwardTabulateSLRPs
        ...
11:     Select and remove an edge $\langle s_p, d_1\rangle \xrightarrow{\pi} \langle n, d_2\rangle$ from WorkList
        ...
22:     foreach $\langle c, d_4\rangle \in$ Incoming $[\langle s_p, d_1\rangle]$ do
          ... //unchanged handling of balanced return flows
31:     od
31.1:   if $d_1 == \mathbf{0} \land$ Incoming $[\langle s_p, d_1\rangle] = \emptyset$ then
31.2:     foreach $c \in$ callSitesCalling(procOf($s_p$)) do
31.3:       foreach $d_5 \in$ returnVal($\langle e_p, d_2\rangle, \langle c, d_1\rangle$) do
31.4:         Propagate($\langle s_{procOf(c)}, \mathbf{0}\rangle \xrightarrow{c} \langle returnSite(c), d_5\rangle$)
31.5:       od
31.6:     od
31.7:   end if
        ...
      end procedure
```

Figure 4: Extension to Support Unbalanced Return Flows (line numbers match the complete representation of the IFDS algorithm shown in Figure 4 of [17])



(a)

```
c(x) {
  y = b(x)
  if(...)
l1:   d(y)
  else
l2:   z = d(const)
  return z
}
```

(b)

Figure 5: Example Flow from Inner to Outer Layer

returns, making unbalanced returns impossible. But FlowTwist starts the analysis on the inside of the API, which naturally calls for supporting such unbalanced return flows.

Figure 4 shows our extension to the IFDS algorithm to enable unbalanced return flows. In line 22 through 31, the algorithm originally loops over all incoming edges and propagates return flows accordingly. Our modification adds lines 31.1 through 31.7. In line 31.1 we check, if the algorithm is currently in a unbalanced situation. This is the case when d_1 is the autological fact $\mathbf{0}$, which always holds, and when there is no incoming edge into d_1 for the current function's starting point s_p. If identified to be in an unbalanced situation, the algorithm computes and propagates return flows to each possible call site. Note that in situations where FlowTwist returns from a method m in an unbalanced way and then processes a call to m again, say with a fact d_1, then this will lead to a situation where the incoming-set of $\langle s_p, d_1\rangle$ is not empty, which is why in this case FlowTwist will perform a normal balanced return, maintaining context sensitivity, returning only to the appropriate call site.

3.3 Creating and Matching Semi-Paths

If working in a regular outside-in manner, a taint analysis would start at sources on the outer level of the API, through some sensitive method such as `Class.forName(..)` and then then back to the original method at which the analysis started. That way, the analysis can report a possible flow as soon as it reaches this starting point again. FlowTwist cannot adopt the same strategy as both its analyses – the one for the integrity problem and the one for the confidentiality problem – are inside-out analyses: They start at some inner layer of the program being analyzed and should report potential flows reaching some outer layer.

To report a flow for the inside-out analyses two conditions have to be met: (1) the flow must reach a source method; we do not define what characterizes source methods at this point, as this depends on the concrete analysis problem addressed. (2) That function must be a transitive caller of the sink.

The need for condition (2) is illustrated in Figure 5a. Assume we start an inside-out analysis at function d. This function returns unbalanced to function c. Function c calls b, from which the flow returns balanced (context-sensitively). Subsequently, the flow returns unbalanced to a. Condition (1) holds for functions a and b. Yet, reporting at b does not make sense, because if b gets called by untrusted code there is no program flow to d as b will return to the untrusted code. This is where condition (2) comes to play. It only holds for a and c, but not for b. So, a flow is only reported at a. Note that condition (2) can be easily checked: It will hold if and only if the function is entered by an unbalanced return.

Once a flow is reported, the algorithm traverses the predecessor chain of facts to construct a semi-path through the program along which a flow exists. It is a semi-path because it only contains one way from a source to a sink. To construct complete paths the semi-paths produced by the two analyses in isolation need to be matched. This has to happen with awareness of context, as it otherwise leads to paths that are infeasible at run time. This context is the call stack, which has to be same to match semi-paths. For the example flow illustrated in Figure 5a, the call stack would be $[a, c, d]$.

Instead of function names, however, FlowTwist uses the concrete call sites to encode the call stack. The code of function c shown in Figure 5b illustrates why concrete call sites are required. In that example, semi-paths exist both for the integrity problem and the confidentiality problem, and they also share the same functions on their call stacks. But, the semi-path for integrity uses the call site labeled as `l1`, while the semi-path for confidentiality uses `l2`. Thus, the two semi-paths should actually not be combined into one execution path. However, this conclusion could not be reached, if FlowTwist used function names rather than concrete call sites in the call stacks.

The question is how to retrieve concrete call sites along the semi-paths. Using the relation *source* does not serve the purpose, because return edges start at an exit statement and end at a return site; the call sites themselves are not included. It is also infeasible to model the call stack as part of facts, because this would make facts caller dependent, thus it is not possible to reuse summary edges across multiple callers. To address the need of storing the call sites we extend our model of facts by the following relations. For facts propagated along call and return edge, *relatedCallSite* maps to the related call site and *callStackEffect* is used to store the effect on the simulated call stack, when traversing a fact. For intra-procedural flow edges, *callStackEffect* maps to *None*.

$$\begin{aligned} relatedCallSite &: Fact \to Statement \\ callStackEffect &: Fact \to [\text{None} \mid \text{Push} \mid \text{Pop}] \end{aligned}$$

The construction of all semi-paths for a fact f is implemented by the work list algorithm shown in Figure 6. As semi-paths are constructed by traversing the predecessor chain of facts, their corresponding call stacks are computed as well. Due to merging facts in the IFDS-algorithm step, the predecessor and neighbor references allow traversing the fact chain in context-insensitive ways. By simulating the call stack in parallel for each semi-path the algorithm ensures that constructed semi-paths only return to callers through which they entered a method (line 23).

Once all semi-paths are constructed, pairwise matches according to their respective call stacks are built and the reversed semi-path of the confidentiality sub-analysis is appended to the semi-path of the integrity sub-analysis. The two thus concatenated semi-paths form a complete path:

$$\begin{aligned}\text{Paths} \;=\; & \{[f_1 \dots f_n, g_m \dots g_1] : \\ & \langle [f_1 \dots f_n], cs_i \rangle \in \text{Integrity-SemiPaths} \wedge \\ & \langle [g_1 \dots g_m], cs_c \rangle \in \text{Confidentiality-SemiPaths} \wedge \\ & cs_i = cs_c \}\end{aligned}$$

3.4 Simplifications to Improve Scalability

In several places we have mentioned the importance of being able to merge similar facts and reuse summary edges to allow the IFDS algorithm to be scalable. Moreover, the IFDS algorithm would not scale if we formulated it in a way that all possible data flows are considered in isolation. However, this is what we do in the construction of semi-path and thus also this step would not scale without some simplifications. The first simplification was already presented implicitly, as only paths are constructed for which it is known that there is a data flow, i.e., by only traversing the data-flow facts generated by the IFDS algorithm. But, our experiments have shown that this is not enough.

A second simplification is to include into a semi-path only those facts f, where $value(f)$ points to a different variable than its successor, meaning the semi-paths will only include facts and statements at which the tainted variable is assigned to another variable, used as argument of a call or being returned (line 9 in Figure 6).

This reduces the number of semi-paths that have to be constructed, as branches not using a tainted variable do not result in additional semi-paths. We think this simplification to be useful also from a usability perspective, as it discards facts in reported paths that are not necessary to comprehend the reported data flow. Note that this simplification is encapsulated in the implementation of *firstIntroductionOf* and can therefore be easily loosened or tightened, e.g., through an implementation that returns only facts at interprocedural edges.

A third simplification is an additional cycle-elimination criterion supplementing the natural elimination criterion that does not include the same facts twice in a semi-path (line 10). During experiments we found huge sub-type hierarchies, which recursively call themselves (e.g. implementations of the decorator pattern). In many cases, precise points-to information is missing, causing conservative approximations to assume call edges to all sub-types. Data flows through such hierarchies result in a combinatorial explosion during semi-path construction. This is because the algorithm will consider each possible sorting order in which the sub-types can call each other.

The cycle-elimination criterion to not include the same fact twice does not help here, as it only prevents including the same function of the same sub-type multiple times. Therefore, we introduce an additional criterion preventing the inclusion of semi-paths calling a function recursively multiple times. This criterion is reflected in the algorithm using the variable *cf* denoting called functions.

3.5 Dependent Analyses

As previously discussed, the enumeration of all possible semi-paths is a critical threat to FlowTwist's scalability. In the following we show how both sub-analysis can be made dependent on each other, such that flows will only be reported if they exist for both the integrity and confidentiality problem, in the same consistent calling contexts. This avoids enumerating semi-paths for which no matching counterpart will exist anyway. As pointed out in former sections, semi-paths of both sub-analyses will only match if the call stack is equal. For this matching only unbalanced return edges are relevant as only these are on the reconstructed call stacks. Balanced return edges will be pushed on the stack also, but in contrast to unbalanced returns these will be popped off the stack again when processing call edges. This behavior is exploited by synchronizing the two sub-analyses on their unbalanced returns, i.e., either analysis should not perform an unbalanced return until the other sub-analysis would return to the same context as well.

Implementing this idea requires a further addition to the IFDS algorithm. All analysis facts are augmented, encapsulating them in a tuple $Fact \times Statement$, whereas the first element is the replaced fact and the second a call site used for the synchronization. Flow functions themselves remain unchanged. They receive only the first element of the tuple as argument. Subsequently, tuples are generated from facts returned by the flow function:

$$wrappedFlow(\langle n, \langle f, s \rangle \rangle) = \{\langle d, s \rangle : d \in flow(\langle n, f \rangle)\}$$

where *flow* is an arbitrary flow function. When seeding initial facts as starting point for the analyses, the statement of the tuple is set to the sink. On unbalanced returns, this statement is replaced by the call site related to that return edge. Importantly, though unbalanced returns are not propagated immediately, unless the other sub-analysis has also reached an unbalanced return with a tuple referencing the same call statement. If there is yet not such a return, the current sub-analysis will pause the return edge, parking it in an internal work list. Paused edges are resumed by the other sub-analysis if encountering the same return, or simply never in case the same return is never reached. In the latter case, this means that there is only an integrity or confidentiality problem, but not both. The extension to the IFDS algorithm for this is shown in Figure 7 and replaces the extension for unbalanced returns shown in Figure 4. The internal work list of each solver is called *leaks*. The algorithm terminates once the work lists of both sub-analysis are empty, disregarding the existence of paused edges.

4. EVALUATION

We performed experiments to compare the proposed inside-out analysis approach with a pure forward analysis in terms of required memory and execution time.

```
    procedure computeSemiPaths(f)
1:  declare WorkList : {[Fact]×[CallSite]×{Function}}
2:  declare SemiPaths : {[Fact]×[CallSite]}
3:  Insert $\langle [f], \emptyset, \emptyset \rangle$ into WorkList
4:  while Worklist $\neq \emptyset$ do
5:    Select and remove an item $\langle [f_1 \dots f_n], [cs_1 \dots cs_m], cf \rangle$ from WorkList
6:    if $f_n = 0$ then
7:      Insert $\langle [f_1 \dots f_{n-1}], [cs_1 \dots cs_m] \rangle$ into SemiPaths
8:    else
9:      foreach $p \in$ firstIntroductionOf($f_n$) do
10:       $valid := p \notin [f_1 \dots f_n]$
11:       if $callStackEffect(p)$ = None then
12:         $cs := [cs_1 \dots cs_m]$
13:       else
14:         $rcs := relatedCallSite(p)$
15:         $decls := initialDeclarations(calledFuncs(rcs))$
16:         switch $e := callStackEffect(p)$ do
17:           case $e$ = Push
18:             $cs := [cs_1 \dots cs_m, rcs]$
19:             $valid := valid \land ((cf \cap decls) = \emptyset)$
20:             $cf := cf \cup decls$
21:           case $e$ = Pop
22:             $cs := [cs_1 \dots cs_{m-1}]$
23:             $valid := valid \land (cs_m = rcs)$
24:             $cf := cf \setminus decls$
25:         end switch
26:       end if
27:       if $valid$ then
28:         Insert $\langle [f_1 \dots f_n, p], cs, cf \rangle$ into WorkList
29:       end if
30:     od
31:   end if
32: od
33: return SemiPaths
    end procedure

    procedure firstIntroductionOf(f)
34: declare WorkList : {Fact}
35: declare Result : {Fact}
36: declare Visited : {Fact}
37: WorkList := $neighbors(f_n) \cup f_n$
38: while Worklist $\neq \emptyset$ do
39:   Select and remove an item $g$ from WorkList
40:   $p := predecessor(g)$
41:   if $p = \emptyset \lor value(g) \neq value(p)$ then
42:     Insert $g$ into Result
43:   else
44:     foreach $n \in neighbors(p) \cup p$ do
45:       if $n \notin$ Visited then
46:         Visited := Visited $\cup\ n$
47:         WorkList := WorkList $\cup\ n$
48:       end if
49:     od
50:   end if
51: od
52: return Result
    end procedure
```

Figure 6: Algorithm to Compute Semi-Paths

```
       declare leaks : {Statement}
       declare paused : {Statement×PathEdge}
       procedure ForwardTabulateSLRPs
         ...
11:      Select and remove an edge $\langle s_p, \langle d_1, s \rangle \rangle \xrightarrow{\pi} \langle n, \langle d_2, s \rangle \rangle$ from WorkList
         ...
31.1:    if $d_1 == \mathbf{0} \land$ Incoming $[\langle s_p, d_1 \rangle] = \emptyset$ then
31.2:      foreach $c \in$ callSitesCalling(procOf($s_p$)) do
31.3:        foreach $d_5 \in$ returnVal($\langle e_p, d_2 \rangle, \langle c, d_1 \rangle$) do
31.4:          $leaks := leaks \cup s$
31.5:          $edge :=$
                   $\langle s_{procOf(c)}, \langle \mathbf{0}, c \rangle \rangle \xrightarrow{c} \langle returnSite(c), \langle d_5, c \rangle \rangle$
31.6:          if $s \in$ otherAnalysis.$leaks$ then
31.7:            otherAnalysis.resume($s$)
31.8:            Propagate($edge$)
31.9:          else
31.10:           $paused := paused \cup \langle s, edge \rangle$
31.11:         end if
31.12:       od
31.13:     od
31.14:   end if
         ...
       end procedure

       procedure resume($s$)
40:      foreach $\langle s', edge \rangle \in paused : s' = s$ do
41:        Propagate($edge$)
42:        $paused := paused \setminus \langle s', edge \rangle$
43:      od
       end procedure
```

Figure 7: Extension to the IFDS Algorithm Making two Analysis Dependent on Each Other

Specifically, the experiments address two research questions under the assumption that there are more sources than sinks:

RQ1: Does the inside-out analysis scale better in terms of memory requirements than a pure forward analysis?

RQ2: Is the inside-out analysis faster than a pure forward analysis?

4.1 Setup

We apply the two versions of the proposed inside-out analysis - with independent and dependent sub-analyses - and a pure forward analysis - the baseline - to the problem of confused deputies in the Java Class Library (JCL) of Oracle Java 7 Update 25. We use two setups for the experiments.

The first setup focuses on call sites of the JCL method `Class.forName(String)`, for which confused deputy attacks (e.g. CVE-2012-4681 and CVE-2013-0422) have occurred in the past. Untrusted code may call `Class.forName`, but is not allowed to retrieve references to classes located in restricted packages, e.g., `sun.*`. Therefore, `forName` checks the permission of its immediate caller. If the caller is unprivileged untrusted code that tries to retrieve a reference to a restricted class an exception is thrown; otherwise, the class reference is returned. JCL classes are trusted, thus privileged code. As such, they may retrieve restricted class references, but must neither leak parameters to `forName` nor return references to classes in restricted class, or must perform permission checks.

We use in total 134 call sites of `forName` as sinks[4]. Sources are parameters to all methods callable by untrusted code, i.e., methods that are either public or protected and declared in a non-final public class not inside a restricted package[5]. For the baseline analysis, we further restrict sources to parameters of type `String` passed to methods that do transitively call a sink. This reduces the number of sources for this analysis to 2,306. This reduction of considered sources is necessary, as otherwise the pure forward analysis would not succeed within up to six hours.

The first experiment setup considers a rather small number of sinks. The second setup considers significantly more sinks. As a reaction to the exploited vulnerabilities due to incorrect use of `forName`, Oracle introduced the annotation `@CallerSensitive` to annotate methods performing permission checks of their immediate caller. In total, there are 89 such methods. The goal is to make it explicit that callers may have to perform checks on their own, but also to replace some manually maintained method white lists. In the experiment, we consider the subset of `@CallerSensitive` methods that may be subject to both an integrity and confidentiality problem, i.e., those that have a receiver or parameters and a return value. There are 64 such `@CallerSensitive` methods, which results in a total of 3,656 call sites considered as sinks.

Both versions of the inside-out analysis and the baseline use the same data-flow facts, flow functions, and construct paths for precise reporting. The flow functions represent normal behavior of a Java program, e.g., handling for assignments, calls and returns. Additional flow functions are implemented to model the behavior of `StringBuilder` and `StringBuffer`. These types are used by developers, but also by the compiler, for string concatenation, which is often used in the problem we focus on in the experiments. If a field is tainted, we conservatively assume this field to be tainted on all instances as we do not apply a precise alias analysis. We do not evaluate conditions and array indices; we always taint the whole array and therefore never kill a taint if only a single array element is overwritten. The algorithm used to construct a call graph is the default algorithm in Soot for whole program analysis, which is an implementation of the Class Hierarchy Analysis (CHA) [3]. As both the inside-out and the pure forward analyses share their flow function definitions, they report the same results and are equally precise.

All analyses of the experiments were executed on a machine with a 4-core Intel(R) Xeon(R) X5560 CPU running at 2.80GHz and 32 GB of RAM. The used operating system is Debian squeeze version 6.0.6 running a 2.6.32-5-xen-amd64 kernel. Each analysis was executed in a fresh JVM process. Heros [1] uses memory-sensitive caches which on the one hand allows it to complete analysis runs even within a restricted amount of memory, but on the other hand causes its analysis time to be heavily dependent on this amount. We thus measured the runtime behavior with different configurations, decreasing the maximum heap size available for the Java VM (-Xmx) by 1 GB starting at 10 GB until we encountered `OutOfMemoryErrors`. For each memory setting we ran five analyses in a row and average their results by use of the arithmetic mean. If an analysis does not succeed in six hours we abort it.

[4]We filtered call sites of `Class.forName` that immediately use constant parameters. Such constructs occur frequently in parts of the JCL as in earlier days static references to the `Class` object of a class were provided using by such a call instead of using the later introduced `class` constant.

[5]The definition of what may be callable by untrusted code is an approximation and should be refined for experiments focusing on precision and recall of the analysis.

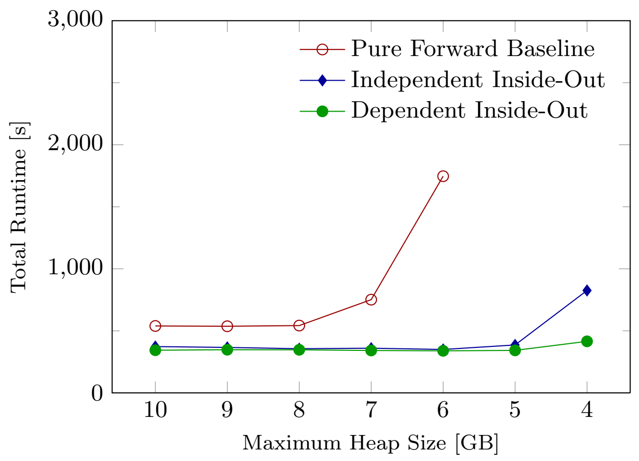


Figure 8: Mean Runtime over Maximum Heap Size in the `Class.forName` Experiment Setup

4.2 Results

The results of the first experiment using call sites of `Class.forName` are shown in Figure 8. All approaches terminate successfully for heap sizes of at least 6 GB. The pure forward baseline encounters `OutOfMemoryErrors` for heap sizes of 5 GB and less, while the inside-out analyses still terminate successfully. For heap sizes of smaller that 3 GB Soot fails to generate an Interprocedural Control Flow Graph and quits with an `OutOfMemoryError`, precluding the analyses from completing.

The runtime of all analyses starts to increase significantly, when approaching the minimum required heap size. For a heap size of 10 GB, the runtime of the baseline analysis is 170 seconds higher than for the independent inside-out approach and 200 seconds higher than for the dependent inside-out approach. The difference increases to a 5 times larger runtime at 6 GB heap size for the baseline.

In Figure 9 the runtime of the analyses is shown for each performed step, whereby *path creation* denotes the semi-path creation and combination into complete paths. The three plots on the left show results for the first experiment setup. As expected the initialization step has equal runtimes in all analyses as their design has no effect on that step. Also most of the time is consumed by the initialization for larger heap sizes. The IFDS step consumes significantly more runtime than the path creation. For the dependent inside-out analysis the values are too small (around 270ms) to appear in the presented plot. However, we encountered in this experiment only a rather small number of resulting semi-paths, that needed to be constructed and matched. We expect the path creation step to become a scalability problem, when more semi-paths result from the IFDS step (c.f. Sections 3.4, 3.5). In the results of the second experiment this will become apparent.

The results for the second experiment using call sites of `@CallerSensitive` annotated methods are shown on the right-hand side of Figure 9. Only the independent and

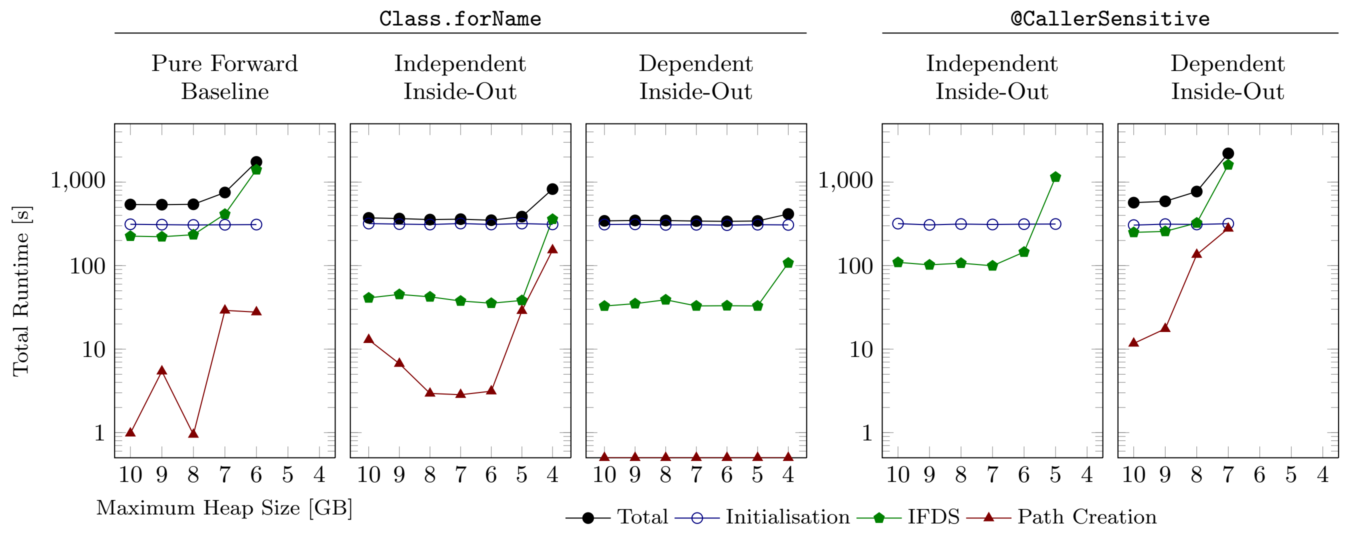


Figure 9: Runtime of Different Analysis Steps

dependent inside-out analyses are shown, because the pure forward baseline analysis was not able to terminate in 6 hours for a heap size of 10 GB; even the IFDS step did not terminate within that time. On the contrary, the independent inside-out analysis is able to perform the IFDS step in roughly 100 seconds, but it fails to compute all semi-paths for all tested heap sizes. Hence, no data for path construction and total analysis runtime can be given in the plot for this analysis. This result confirms our assumption that semi-path construction does pose a bottleneck. Only the dependent inside-out analysis terminates successfully in 570 seconds of total runtime at a heap size of 10 GB. The minimum heap size requirement for the dependent inside-out analysis is also larger than for the first experiment setup.

To conclude, the results of the first experiment already indicated the answer to both research questions. However, the second more realistic experiment setup gives a clear answer: the inside-out analysis approach scales better and performs faster than the pure forward baseline analysis.

5. RELATED WORK

We first compare to other static analyses for detecting vulnerabilities that allows for confused-deputy or collusion attacks. We further compare to taint analyses in general as well as information-flow analyses, and comment on popular extensions to IFDS.

5.1 Confused-Deputy and Collusion Attacks

While our analysis approach is the first to address confused-deputy problems in Java, the problem has been widely discussed in the setting of Android security. In Android, apps are given capabilities by assigning them static permissions at installation time. After having been granted a capability, an app must take care not to expose this capability through APIs that might be callable from unauthorized third-party apps. Exposing such APIs causes a confused-deputy problem. In situations in which the API is exposed on purpose, one speaks of a “collusion attack” in which both the caller and the callee app conspire against the user, for instance to leak contact information to the internet, with the caller app having the contact permission and the callee app having the internet permission only.

Woodpecker [8] is a tool-based approach for finding accidental capability leaks in Android applications, particularly tuned towards pre-installed apps on stock smartphones. It identifies capability leaks as paths from an app’s public API to certain sensitive Android-API methods. A leak is reported if the path includes no permission checks. Woodpecker implements a forward analysis only starting at all of the app’s entry points; it does not check if the values returned from a sensitive low-level API are actually returned to the caller. The tool is not implemented within a program-analysis framework but rather as a “mixture of Java code, shell scripts and Python scripts” over an off-the-shelf disassembler. Due to this design, Woodpecker is context-insensitive.

CHEX [15] is an approach with a similar goal to Woodpecker but is implemented on top of the Watson Libraries for Analysis (WALA) [23], which allows it to conduct a context-sensitive analysis (0-1-CFA). CHEX further includes an advanced modeling of the Android execution lifecycle, which is important to gain recall. As Woodpecker, also CHEX performs a forward analysis only, without tracking return values of sensitive APIs.

Zhou and Jiang [24] developed an approach to find unprotected content providers in Android apps. Malware apps can misuse such content providers to steal or modify data managed by the vulnerable app. As such, their tool ContentScope also needs to determine both an integrity problem (to identify potential for data modification) and a confidentiality problem (to identify data leakage). Interestingly, the details given on the analysis suggest, though, that ContentScope only tracks malicious input to the content providers’ low level APIs but does not, in fact, check whether leaked values are returned back to the attacker. For content providers this might actually be sufficient because their API methods are mean to return data objects—this is their very purpose. In Java, with methods such as `Class.forName`, the situation is entirely different: here many returned Class objects may not actually leak to malicious callers.

Marfori et al. [16] try to assess the gravity of the problem of collusion attacks in the Android space by developing an approach that allows researchers to assess the potential for such attacks on a large scale. The approach over-approximates the potential by analyzing static permissions and direct API calls only; it implements no data-flow analysis.

Octeau et al. [18] developed Epicc, a static-analysis tool to detect the targets of inter-component communication (ICC) calls in Android, for instance using the popular "Intent" API. Epicc can be used to resolve and match ICC calls in general, allowing researchers to determine which apps can call one another, and with which messages. Analyses for collusion attacks can build on Epicc's results. Epicc mainly consists of a string analysis and does not track flows of attacker-controlled or private data.

Bugiel et al. [2] developed a system to detect and mitigate Android collusion attacks at runtime.

5.2 Taint Analysis

FlowTwist implements a special form of taint analysis. Many taint analyses have been developed over the past few years, focusing on different programming languages and security-sensitive APIs. We here focus on approaches for Java and Android. All analyses presented track flows forward from a given set of sources in an attempt to find a path to a set of sinks.

The static taint analysis tool TAJ (Taint Analysis for Java) [22] is implemented in WALA [23] and focuses on web applications. As part of a commercial product it possesses a certain degree of maturity: For instance, it scales to large applications by using a priority-driven call-graph construction which provides intermediate partial results based on a priority function. Tripp et al. specifically adapted the tool to analyze Java EE applications; hence, it is able to handle Java beans and frameworks configured by XML files. Further optimizations of the runtime include the parsing of the artifacts that are used as source to generate code instead of analyzing the generated code. WALA also supports unbalanced analysis problems, however supports only forward analyses in general. To the best of our knowledge, the solution to unbalanced analysis problems has never before been formalized.

Andromeda [21], another tool from Tripp et al. used in a commercial product, is also a static taint analysis for web applications. Because alias analysis and even (partial) call-graph generation are invoked on demand, it is very scalable. It utilizes Framework For Frameworks (F4F) [20], a taint analysis specifically designed for frameworks like Apache Struts or Spring. Additionally Andromeda is capable of performing incremental analyses on updated web applications. For resolving aliases, it uses a context-sensitive on-demand alias analysis.

FlowDroid [5] is the currently most precise taint analysis for Android. To improve recall, it thoroughly models Android's execution lifecycle. To obtain precision, it uses a fully context and flow sensitive formulation within the IFDS framework, along with an on-demand pointer analysis inspired by the one of Andromeda.

5.3 Information-Flow Analysis

Information-flow analysis distinguishes intself from taint analysis by also tracking implicit information flows that can occur through control flows such as conditional branches.

Genaim et al. [6] claim to have implemented the first information flow analysis for Java byte code. It is implemented with the static analyzer Julia [13]. The taint information propagated consists of a single boolean value which is very lightweight, but not sufficient for many fields of application. Their approach is able to detect implicit flows in loops and exceptions while preserving flow- and context-sensitivity. All fields are treated as static class variables, making the approach field based. No information is given on how the approach deals with aliasing.

Hammer et al.[10] present a flow-, context- and object-sensitive information flow analysis for Java applications based on program dependence graphs. JOANA (Java Object-sensitive ANAlysis) [9] is an evolution of Hammer's analysis. It has recently been extended to deal with possibilistic and probabilistic leaks in concurrent Java programs [7].

6. CONCLUSION

We have presented FlowTwist, a novel approach to taint-analysis addressing integrity and confidentiality issues at the same time. FlowTwist exploits the idea of reversing the analysis problem, which is known to be beneficial when more sources than sinks are given in a system. Moreover, this work presents how to exploit this characteristic, when addressing integrity and confidentiality issues at the same time. FlowTwist instantiates the IFDS algorithm with a backward and a forward taint analysis, both of which, operate inside-out, from inner layers of the API to attacker-callable functions at the outer API layer. We further explained how FlowTwist extends the IFDS algorithm to maintain context sensitivity in this situation and to efficiently construct paths which give useful information to the developer trying to fix a detected issue. In experiments we compared the suggested inside-out approach against a pure forward analysis. Results of these experiments confirmed that an inside-out approach scales better and is faster.

In future work we plan to elaborate on two parts of the proposed approach. First, applying the analysis to more vulnerabilities with a focus on precision and recall. This requires some more work on modelling permission checks and the definition of when a data-flow presents a true integrity and confidentiality issue, which was out of scope for the current work that focuses on the general inside-out idea and its scalability. Second, we want to investigate possibilities of interweaving the path generation and the IFDS algorithm even more, e.g., one idea is to create finite state machines, whereas transactions represent changes to the simulated call stack, such that combined paths can be generated by traversing the same transactions in both state machines resulting from each sub-analysis.

7. ACKNOWLDGEMENTS

We wish to thank Cristina Cifuentes and Andrew Gross from Oracle for their input on our work, as well as all participants of the Dagstuhl Seminar 13162 on Pointer Analysis for the lively discussion about the challenges this project involves. This work was supported by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by the DFG within the project RUNSECURE.

8. REFERENCES

[1] E. Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *1st ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis*, pages 3–8, 2012.

[2] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *19th Annual Network and Distributed System Security Symposium*, 2012.

[3] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101. Springer, 1995.

[4] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 73–84, New York, NY, USA, 2013. ACM.

[5] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for android applications. Technical Report TUD-CS-2013-0113, EC SPRIDE, 2013.

[6] S. Genaim and F. Spoto. Information flow analysis for java bytecode. In *Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation*, pages 346–362, 2005.

[7] D. Giffhorn and G. Snelting. A new algorithm for low-deterministic security. Technical report, Karlsruhe Institute of Technology, 2012.

[8] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[9] J. Graf, M. Hecker, and M. Mohr. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Proceedings of the 6th Working Conference on Programming Languages*, 2013.

[10] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[11] N. Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[12] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth. Slicing droids: Program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1844–1851, New York, NY, USA, 2013. ACM.

[13] Julia. `http://www.juliasoft.com/products`, retrieved 2014-03-16.

[14] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[15] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240, 2012.

[16] C. Marforio, A. Francillon, S. Capkun, S. Capkun, and S. Capkun. *Application collusion attack on the permission-based security model and its implications for modern smartphone systems.* Department of Computer Science, ETH Zurich, 2011.

[17] N. A. Naeem, O. Lhoták, and J. Rodriguez. Practical extensions to the IFDS algorithm. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, pages 124–144, 2010.

[18] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *USENIX Security Symposium 2013*, 2013.

[19] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[20] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 1053–1068, 2011.

[21] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, 2013.

[22] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 87–97, 2009.

[23] T. J.Watson Libraries for Analysis (WALA). `http://wala.sf.net/`, retrieved 2014-03-16.

[24] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Annual Symposium on Network and Distributed System Security*, 2013.