Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis With Unbounded Access Paths

Johannes Lerch*, Johannes Späth†, Eric Bodden*† and Mira Mezini*‡
*Technische Universität Darmstadt, †Fraunhofer SIT, ‡Lancaster University
*†Darmstadt, Germany, ‡Lancaster, United Kingdom
*{lastname}@cs.tu-darmstadt.de, †{firstname.lastname}@sit.fraunhofer.de

***Abstract*—Precise data-flow analyses frequently model field accesses through access paths with varying length. While using longer access paths increases precision, their size must be bounded to assure termination, and should anyway be small to enable a scalable analysis.**

We present Access-Path Abstraction, which for the first time combines efficiency with maximal precision. At control-flow merge points Access-Path Abstraction represents all those access paths that are rooted at the same base variable through this base variable only. The full access paths are reconstructed on demand where required. This makes it unnecessary to bound access paths to a fixed maximal length.

Experiments with Stanford SecuriBench and the Java Class Library compare our open-source implementation against a field-based approach and against a field-sensitive approach that uses bounded access paths. The results show that the proposed approach scales as well as a field-based approach, whereas the approach using bounded access paths runs out of memory.

I. Introduction

Static program analyses and especially data-flow analyses usually have to consider values being assigned to and read from local variables and fields. While local variables are often simple to track, the systematic handling of fields can be complex. Techniques for modeling fields can be distinguished into field-based and field-sensitive approaches. Field-based techniques model a field access `a.f` simply by the field's name `f`, plus potentially its declaring type—a coarse grain approach that ignores the base object `a`'s identity. Field-sensitive techniques, on the other hand, include the base variable `a` in the static abstraction, potentially increasing analysis precision as fields belonging to different base objects can be distinguished.

While many existing analyses restrict their field-sensitivity to a single level, more precise analyses represent static information through entire *access paths* – a base variable followed by a finite sequence of field accesses [1]–[3]. For instance, assume a taint analysis determining whether (and where) the example shown in Figure 1a might print the password. Here the analysis must distinguish access paths of length $k \geq 2$ (k is the maximal length of the finite sequence) to determine that a leak can occur only at the first print statement. While the use of access paths can increase precision, one must generally bound k to a finite maximum, as otherwise loops or recursive data structures such as linked lists might cause abstractions such as `l.next.prev.next...` to grow indefinitely, which would cause the analysis not to terminate.

A common approach to deal with infinite chains of field accesses is k-limiting [4], which includes in the abstraction only the first k nested field accesses, abstracting from all others. If fields are read, the analyses frequently assume that any field accessible beyond the first k fields may relate to the tracked information. Hence, k-limiting introduces an over-approximation that becomes less precise for smaller values of k. In contrast, a high k value means the analysis will distinguish more states. In this work, we present experiments, which show that analyses can run out of gigabytes of main memory when analyzing real-world programs even with small k values.

To address the problems raised above, we present Access-Path Abstraction, a novel and generic approach for handling field-sensitive analysis abstractions *without the need for k-limiting*. To keep access paths finite and small, at control-flow merge points Access-Path Abstraction represents all those access paths that are rooted at the same base variable through this base variable only. In a summary-based inter-procedural analysis this leads to fewer and yet highly reusable summaries, which can speed up the analysis significantly: A procedure summary for a base object `a` can represent information for all access paths rooted in `a`. To maintain the precision of field-sensitive analyses, Access-Path Abstraction reconstructs the full access paths on demand where required.

We present Access-Path Abstraction as a novel extension to the IFDS framework for inter-procedural finite distributive subset problems [5] and hereafter use IFDS-APA as an acronym for this extension. We provide an open-source implementation on top of the IFDS/IDE solver Heros [6].[1]

In experiments using Stanford SecuriBench and the Java Class Library as benchmarks, we compare IFDS-APA against a field-based approach and against a field-sensitive approach using k-limiting, both in terms of scalability and analysis time. The results show that IFDS-APA scales as well as a field-based approach, whereas a field-sensitive approach using k-limiting runs out of memory.

To recap, this work presents the following original contributions:

- a novel extension to the IFDS framework enabling scalable and precise field-sensitive analysis without the need for k-limiting,
- a full open-source implementation, and
- extensive experiments comparing the performance of the proposed approach to a field-based and a field-sensitive approach with k-limiting.

[1] Heros is hosted on GitHub: https://github.com/sable/heros Our implementation is available there.

```
a.g = password();
b.f = a;
print(b.f.g);
print(b.f.h);
```

(a)

```
x = source();
A a1 = new A();
A a2 = new A();
a1.f = x;
y = a2.f;
sink(y);
```

(b)

Fig. 1: Precision of Field-Based and Field-Sensitive Models

The remainder of this paper is structured as follows. In Section II we give a short summary of the IFDS framework, while Section III explains common ways to model field accesses. In Section IV we describe our changes and extensions to the IFDS framework. Section V covers the evaluation, while we discuss related work in Section VI and conclude in Section VII.

II. Background on IFDS

Due to its efficiency, many data-flow analyses [2], [7], [8] are implemented as instantiations of the IFDS framework [5]. We next briefly summarize important background information on IFDS, necessary to understand our extension IFDS-APA. The IFDS framework is capable of solving inter-procedural finite distributive subset problems. At every call, the framework computes callee summaries on the fly. These summaries are highly reusable as (1) they are independent of the calling context and (2) due to the distributivity of the analysis problem can be reused on an element-by-element basis. Internally, IFDS transforms the original analysis problem into a reachability problem over the so-called *exploded super graph*.

A definition of an IFDS problem consists of a *data-flow domain* D and a set of *flow functions*. The data-flow domain is a set of *data-flow facts*. A solution to the IFDS problem provides information whether a given fact holds at a certain statement. One special fact is the **0**-fact, a tautological fact that always holds. Other facts can be generated unconditionally by deriving them from this **0**-fact. To guarantee termination, IFDS requires the data-flow domain to be a finite set. To express an IFDS problem, one defines the flow functions which describe how data-flow facts are transferred from one statement to its (intra and inter-procedural) successors. A requirement for IFDS is that each flow function, say f, needs to be distributive w.r.t. set union, thus $f(A \cup B) = f(A) \cup f(B)$ for $A, B \subseteq D$. This allows the framework to define flow functions on single elements of D. The evaluation of the flow function for any such element yields a subset $S \subseteq D$.

A. Path Edges

IFDS computes a callee's procedure summary incrementally through so-called *path edges*. A path edge is written as $\langle s, d_1 \rangle \rightarrow \langle t, d_2 \rangle$, where t is an arbitrary *target statement* and s is the *start statement* of the method of t. This makes a path edge always local to one method. The elements d_1 and d_2 are data-flow facts: d_2 is the *target fact* and d_1 the *start fact* of the edge. Semantically, a path edge expresses: If d_1 is reachable (and therefore holds) at statement s, so is d_2 at statement t. The purpose of the flow functions is to successively deduce new path edges to bridge longer and longer paths. The appropriate flow function for the target statement t, say f_t, receives the data-flow fact d_2 as arguments. The result of the application of the flow function is a set of data-flow facts $S = f_t(d_2)$. For each element d_3 of the set S and each statement t' succeeding t a path edge is derived: $\langle s, d_1 \rangle \rightarrow \langle t', d_3 \rangle$.

B. Incoming Set And Summaries

An important part of the IFDS framework is its *incoming set*. For each method and data-flow fact which enters the method through a call site, the incoming set stores the information (1) through which call site(s) the method is entered and (2) which path edges reach those call site(s). Consider a method m is called at a call site c and the path edge $p = \langle s, d_0 \rangle \rightarrow \langle c, d_1 \rangle$ reaches c. Further assume, that the call's flow function maps fact d_1 to d_2 on the side of the callee m. Once entering the method, the path edge p is added to the incoming set $I_m^{d_2}$ for the callee method m and fact d_2, so $p \in I_m^{d_2}$.

IFDS seeks to construct procedure summaries that are independent of any particular calling context. It thus bootstraps the analysis of any callee m by propagating an initial self loop edge $\langle v, d_2 \rangle \rightarrow \langle v, d_2 \rangle$ from the start statement v of m. This expresses that d_2 holds at v if d_2 holds at v, i.e., without any further condition. From this edge, new path edges are successively derived via the application of the flow-functions. The flow-functions maintain the start statement and start node of the derived edges. Whenever a derived path edge, say $\langle v, d_2 \rangle \rightarrow \langle t, d_3 \rangle$, reaches an exit statement t of the method m, this path edge becomes a (partial) procedure summary. The framework must now apply this summary's effect to all callers of m. It traverses the incoming set $I_m^{d_2}$ to extract all call sites at which path edges inside callers have to be continued. A return-flow function maps d_3 back to a fact d_4 in the caller scope. Here, assuming the path edge p is in the incoming set, IFDS continues with the path edge $\langle s, d_0 \rangle \rightarrow \langle c', d_4 \rangle$ for any successor statement c' of c.

The path edge $\langle v, d_2 \rangle \rightarrow \langle t, d_3 \rangle$ is stored as a intra-procedural summary for method m. The summary can be re-applied for any other call site which is interested in analyzing the method m with the same input fact d_2.

III. Modeling Fields in Data-Flow Analysis

In this section, we discuss two alternatives for modeling field accesses in data-flow analyses. We use taint analysis as an example client, however, the described techniques are applicable to data-flow analyses in general. A taint analysis reasons about possible data flows from a given source to a given sink and can decide privacy as well as integrity problems [6].

A. Field-Based Models

Field-based analyses treat fields independently of the objects they belong to. They track a field as soon as a tracked value is assigned to it, independent of the object instance the field belongs to. Thus, any subsequent read from the field must be tracked no matter which object the field belongs to. A possible analysis domain D for a field-based analysis comprises all local variables $\mathcal{L}$ of all program's methods and all fields $\mathcal{F}$ declared in the program, so $D = \mathcal{L} \cup \mathcal{F}$. While a field-based analysis can be sound, not considering the base object will often lead to imprecision, as illustrated by the following example.

```
foo(){
  A a = new A();
  a.f = source();
  A b = id(a);
}
bar(){
  A a = new A();
  a.g = source();
  A b = id(a);
}
id(A p){
  return p;
}
```

(a) Multiple Summaries

```
foo(A a) {
  while(unknown()) {
    A b = new A();
    if(unknown())
      b.f = a;
    else
      b.g = a;
    a = b;
    b = null;
  }
  return a;
}
```

(b) State Explosion

Fig. 2: Examples of Threats to Scalability

Example 1: In Figure 1b, field `a1.f` is assigned the tainted value of `x`. Field `a2.f` never gets tainted. Nevertheless, the analysis will report it as tainted, because it models both field accesses as `A.f`, resulting in a false positive.

B. Field-Sensitive Models

A more precise alternative is to model field accesses as *access paths*. An access path consists of a base variable – a local variable visible in the current method's scope (including its parameters and the receiver `this`) – followed by a sequence of field accesses. In a taint analysis, an access path typically models an access path through which a tainted memory location can be reached. Field-sensitive models are more precise than field-based ones. In Figure 1b a field-sensitive analysis would taint the access path `a1.f` but not `a2.f`. When processing the read `y = a2.f`, no taint will be reported for `y`, avoiding a false warning.

Unfortunately, the described data-flow domain is unbounded. Assume a further assignment `a3.g = a1` somewhere in the code in Figure 1b. To maintain precision the analysis must propagate the taint from `a1.f` to `a3.g.f`, resulting in an access path of length 2. If proper care is not taken, loops can yield access paths of an unbounded length. Also analyzing recursive data structures, e.g., doubly-linked lists, may yield unbounded access paths such as `l.next.prev.next...`. Although `l.next` and `l.next.prev.next` statically refer to the same object, most analyses are unable to identify this equivalence.

Formally, for a field-sensitive taint analysis the data-flow domain is identifiable as the set

$$D := \{(x_1, \ldots, x_k) \mid k \in \mathbb{N}, x_1 \in \mathcal{L}, x_i \in \mathcal{F}, \forall i \geq 2\}.$$

This domain of infinite size contradicts the requirements of IFDS and other data-flow analysis frameworks that guarantee termination only for finite data-flow domains.

To obtain a finite domain, it is common practice to artificially bind the sequence of field accesses to a fixed length by limiting k in the domain definition to a given natural number. This is known as *k-limiting* [4]. Analyses then have to suitably alter the processing of bounded access paths to retain soundness. Limiting k obviously introduces an over-approximation. Decreasing the selected k value increases the over-approximation and lowers the precision. Increasing the k value enables the analysis to distinguish more states, however, at the cost of defeating its scalability due to state explosion. Our experiments—presented in Section V—show that even for small k values analyses can run out of gigabytes of main memory when analyzing real-world programs.

There are two main root causes for the scalability problems: (a) due to the way applicability of method summaries is defined, even methods that represent an identity function w.r.t. an input tainted value must be re-analyzed for each access path rooted at that tainted value, (b) due to an explosion of different states to be considered. The following two examples illustrate the root causes.

Example 2: Methods that represent an identity function w.r.t. the tainted value must be re-analyzed for each possible call site. For illustration, consider the code in Figure 2a. Assume a class `A` with two fields `f` and `g`. Method `foo` taints `a.f`, which a field-sensitive analysis will model by the access path `a.f`. Next, this access path flows as a parameter into method `id`. Typical summary-based analyses will translate the abstraction `a.f` to the scope of the callee, yielding `p.f`. Next, the analysis will create a procedure summary for `id`, indicating that it taints `retVal.f` if `p.f` was tainted. Now, consider the second calling context for `id` within `bar`, which passes a tainted value `a.g`. Again, this value is translated into `p.g`. Since the computed summary for `p.f` is not applicable to `p.g`, the analysis will process the `id`-procedure again, although `id` returns the parameter unchanged. While the analysis effort is trivial in this example, the method `id` could in reality have many more statements and may call many other methods, while remaining an identity function, i.e., opaque, w.r.t. the tainted value. Such methods can be quite frequent in an application.

Example 3: Each unique access path must be propagated resulting in an explosion of the propagated facts. For illustration consider the code in Figure 2b. Assume that the parameter `a` of method `foo` is tainted and that we cannot statically decide the values to which the `while` and `if` conditions will evaluate, i.e., both branches are possible for each condition. Each loop iteration thus propagates from `a` to `b.f` and `b.g`. Hence, `foo` may return tainted access paths `a` (no loop iteration), `a.f` and `a.g` (one iteration), `a.f.f`, `a.f.g`, `a.g.f`, and `a.g.g` (two iterations), and so on. Given a maximum access path length of k and A the set of fields written inside the loop, this yields $\sum_{n=0}^{k} |A|^n$ different access paths. In IFDS the merge operator at control-flow merge points is restricted to set union. Thus, the analysis cannot merge facts and must propagate each unique access path.

In our experiments we observed that constructs similar to those illustrated by this example defeated the scalability of our field-sensitive client analyses. The most common construct was actually not a loop, but methods invoked on an interface type with many concrete implementations. If it is undecidable to which concrete implementation the method call must be resolved, analyses typically assume that any of them may be called. If one of them recursively invokes the same interface method, we observe the same effect as in the example – the loop manifests itself at an inter-procedural level through recursion.

IV. Approach

We propose Access-Path Abstraction, a sound and scalable framework for field-sensitive data-flow analysis. Access-Path Abstraction obviates the need for over-approximations like k-limiting for keeping the domain finite and hence also the need to tradeoff between precision and scalability.

The key ideas underlying Access-Path Abstraction are:

1) Abstract summaries are constructed that abstract over a whole set of access paths, e.g., if a method is invoked with an access path `a.f`, Access-Path Abstraction creates a summary for `a.*` instead.
2) Abstract summaries, such as `a.*` are reused for all callers invoking the respective method with any access path covered by the abstraction, e.g., `a.g` or any other access path sharing the common prefix `a`. Yet, the application of summaries is designed such that the result reflects specific access paths of specific callers.
3) Access-Path Abstraction ensures that only those parts of the application are analyzed that would be analyzed, if concrete access paths were used instead of abstract summaries. For example, a taint analysis processing an assignment `b = a.g` should continue for `b` if and only if `a.g` is tainted, but not because any other tainted field of `a`, e.g., `a.f`, was abstracted to `a.*`.

Next, we describe the changes and extensions to the IFDS framework that are required to achieve 1) and 2), then we elaborate on changes for 3). Subsequently, we discuss how the extensions can be adapted to loops and return sites to solve the state-explosion problem described in Example 3 and why Access-Path Abstraction obviates the need for over-approximations like k-limiting keeping the domain finite. IFDS-APA, the variant of Access-Path Abstraction we describe here, extends the IFDS framework [5].

A. The Analysis Domain

IFDS-APA uses an efficient abstraction that encodes *sets of* access paths in a concise, symbolic representation.

Access Path: Given the set of all local program variables $\mathcal{L}$ and the set of all field declarations $\mathcal{F}$, an *access path* α is a sequence of a local variable followed by a sequence of field accesses: $\alpha = (\alpha_1, \ldots, \alpha_n)$ with $\alpha_1 \in \mathcal{L}$ and $\alpha_i \in \mathcal{F}$ for $2 \leq i \leq n$ and $n = |\alpha|$.

Access-Path Bundles: To realize the idea of constructing abstract summaries 1), we formulate an analysis domain that implicitly encodes *sets of* access paths.[2] In this domain, a fact is called an *access-path bundle* and consists of an access path α and a *set of exclusions* E, $E \subseteq \mathcal{F}$. Exclusions are the key to avoid unnecessary analysis steps due to over-approximation 3), as we explain later. The tuple $\langle\!\langle \alpha, E \rangle\!\rangle$ represents a whole set of access paths, formally defined as:

$$\begin{aligned}\langle\!\langle \alpha, E \rangle\!\rangle &= \langle\!\langle (\alpha_1, \ldots, \alpha_{|\alpha|}), E \rangle\!\rangle \\ &:= \{(x_1, \ldots, x_m) \mid \quad x_i = \alpha_i, 1 \leq i \leq |\alpha|, \\ &\qquad\qquad m \geq |\alpha|, \\ &\qquad\qquad x_{|\alpha|+1} \notin E\}\end{aligned}$$

[2] A client analysis may include specific information into facts. Our implementation anticipates this. However, to ease the exposition, throughout this paper, we restrict facts to just a pair of an access path and the set of exclusions.

Hence, the bundle $\langle\!\langle \alpha, E \rangle\!\rangle$ represents all access paths of length greater or equal to $|\alpha|$, which start with the sequence α, and whose field $x_{|\alpha|+1}$ is not in E, if the length of the path is greater than $|\alpha|$. We say that a bundle $\langle\!\langle \alpha, E \rangle\!\rangle$ belongs to the local $l \in \mathcal{L}$, iff $\alpha_1 = l$. For brevity, we write $\widetilde{\alpha}$ to denote a bundle $\langle\!\langle \alpha, E_\alpha \rangle\!\rangle$.

In the following, we define some operations on access-path bundles, which are used in the process of constructing and applying summaries.

Partial Order on the set of All Bundles: The relation $\langle\!\langle \alpha, E_\alpha \rangle\!\rangle \supseteq \langle\!\langle \beta, E_\beta \rangle\!\rangle$ holds, iff the following conditions (given $\alpha = (\alpha_1, \ldots, \alpha_k)$ and $\beta = (\beta_1, \ldots, \beta_l)$) are fulfilled:

1) $|\alpha| \leq |\beta|$
2) $\alpha_i = \beta_i, \quad 1 \leq i \leq |\alpha|$
3) $\begin{cases} \beta_{|\alpha|+1} \notin E_\alpha, & \text{if } |\alpha| < |\beta| \\ E_\alpha \subseteq E_\beta, & \text{if } |\alpha| = |\beta| \end{cases}$

Intuitively, α is a prefix of β and β is not excluded through the exclusion set E_α, if the length of β is greater than α; if they have the same length, E_β excludes at least the fields excluded by E_α.

If two access-path bundles only differ in their exclusions, we write $\langle\!\langle \alpha, E_\alpha \rangle\!\rangle \approx \langle\!\langle \beta, E_\beta \rangle\!\rangle$, formally defined as:

1) $|\alpha| = |\beta|$
2) $\alpha_i = \beta_i, \quad 1 \leq i \leq |\alpha| = |\beta|$.

For two access-path bundles $\widetilde{\alpha}$ and $\widetilde{\beta}$ such that $\widetilde{\alpha} \approx \widetilde{\beta}$ or $\widetilde{\alpha} \supseteq \widetilde{\beta}$, we define their *delta*, denoted by $\Delta(\widetilde{\alpha}, \widetilde{\beta}) \in (\mathcal{F})^* \times 2^{\mathcal{F}}$ as

$$\Delta(\widetilde{\alpha}, \widetilde{\beta}) := \begin{cases} ((\beta_{|\alpha|+1}, \ldots, \beta_{|\beta|}), E_\beta) & \text{if } |\alpha| < |\beta| \\ (\epsilon, E_\alpha \cup E_\beta) & \text{if } |\alpha| = |\beta| \end{cases}$$

ϵ here refers to the empty sequence. Note that a delta is not an access-path bundle and never contains a local variable.

Concatenation of an Access-Path Bundle and a Delta: Given a bundle $\widetilde{\alpha}$ and an arbitrary delta $\Delta = (\delta, E_\delta)$, we define the concatenation, $\widetilde{\alpha} :: \Delta$, which yields a new bundle, as follows:

$$\widetilde{\alpha} :: \Delta := \begin{cases} \langle\!\langle (\alpha_1, \ldots, \alpha_{|\alpha|}, \delta_1, \ldots, \delta_{|\delta|}), E_\delta \rangle\!\rangle & \text{if } \delta \neq \epsilon \\ \langle\!\langle (\alpha_1, \ldots, \alpha_{|\alpha|}), E_\alpha \cup E_\delta \rangle\!\rangle & \text{if } \delta = \epsilon. \end{cases}$$

If δ is non-empty, then its elements are used to refine the access-path α (by extension). As exclusions always refer to the position after the end of the access path, in this case one retains only the exclusions E_δ. In the other case, both sets of exclusions are retained.

B. Abstract Summaries

At specific statements, called *abstraction points*, IFDS-APA abstracts over a given access-path bundle to compute an abstract summary. Abstraction points are all merge points of intra or inter-procedural control flows: (a) start points of methods, (b) entries to loops, and (c) return sites. In the following, we will focus first on (a) as the most intuitive of those cases.

The key to the efficiency of IFDS-APA is that in cases where an access-path bundle $\widetilde{\alpha} = \langle\!\langle (\alpha_1, \ldots, \alpha_n), E \rangle\!\rangle$ is passed to a callee c, IFDS-APA will bootstrap c's analysis with an

initial self-loop path edge not of $\widetilde{\alpha}$ but of an abstracted version of $\widetilde{\alpha}$ that represents all access paths rooted in the same local variable as $\widetilde{\alpha}$, i.e., $\langle\!\langle(\alpha_1), \varnothing\rangle\!\rangle$. For instance, in a situation, where a callee would normally be analyzed for $\langle\!\langle \texttt{a.f.g}, \varnothing\rangle\!\rangle$, IFDS-APA initiates an analysis for $\langle\!\langle \texttt{a}, \varnothing\rangle\!\rangle$ instead. If a summary for the abstracted bundle already exists, IFDS-APA applies this summary without further computation.

While the application of summaries to avoid computations is equal to IFDS, IFDS-APA requires a different definition of both (i) when a summary is applicable, and (ii) how it is applied. In line with IFDS terminology [5] summaries are of the form $\langle sp_{callee}, d_3\rangle \rightarrow \langle ep_{callee}, d_4\rangle$: "$d_4$ holds at end point ep_{callee} in any context in which d_3 holds at the start point sp_{callee}". A summary $\langle sp_{callee}, d_3\rangle \rightarrow \langle ep_{callee}, d_4\rangle$ is applicable to an incoming fact $d_2 = \langle\!\langle \alpha, E_a\rangle\!\rangle$ iff $d_3 \sqsupseteq d_2$. The summary is applied by computing the delta between d_3 and d_2 and concatenating it to d_4. Thus, for a start point sp_{caller} at the caller for which the fact d_1 resulted in d_2 being propagated to the callee, we generate the path edge $\langle sp_{caller}, d_1\rangle \rightarrow \langle rs, d_4 :: \Delta(d_3, d_2)\rangle$ to the return site rs.[3]

Example 4: Referring back to Example 2 and Figure 2a, IFDS-APA analyzes method `id` only once for the access-path bundle $\langle\!\langle p, \varnothing\rangle\!\rangle$. Method `foo` calls `id` with an incoming fact $\langle\!\langle p.f, \varnothing\rangle\!\rangle$ and method `bar` with $\langle\!\langle p.g, \varnothing\rangle\!\rangle$. The summary for `id` maps $\langle\!\langle p, \varnothing\rangle\!\rangle$ to $\langle\!\langle retVal, \varnothing\rangle\!\rangle$ and is applicable to both incoming facts. The application computes the deltas $(f, \varnothing)$ and $(g, \varnothing)$ and their concatenation to the returned value yields the facts $\langle\!\langle b.f, \varnothing\rangle\!\rangle$ for method `foo` and $\langle\!\langle b.g, \varnothing\rangle\!\rangle$ for method `bar`. In other words, concatenating the deltas restores the calling context that was abstracted away before the callee was analyzed.

C. Field-Read and Field-Write Statements

Next, we explain how IFDS-APA handles cases, where callee procedures read from or write to access paths that were abstracted on entry to the callee.

Field-Read Statements: Assume that we are analyzing a callee c with a taint represented by an abstracted access-path bundle $\langle\!\langle \texttt{x}, \varnothing\rangle\!\rangle$. If c reads `x.f`, the analysis cannot know which of `x`, `x.f`, or `x.g` etc. are tainted, as $\langle\!\langle \texttt{x}, \varnothing\rangle\!\rangle$ effectively represents all access paths `x.*`.

IFDS-APA addresses this problem by computing the required information on the fly. It checks c's incoming set for callers that explicitly requested the analysis with respect to `x.f` before the abstraction occurred: For a path edge $\langle sp, \langle\!\langle(\alpha_1, \ldots, \alpha_n), E_\alpha\rangle\!\rangle\rangle \rightarrow \langle \texttt{b=x.f}, \langle\!\langle x, E_\beta\rangle\!\rangle\rangle$ IFDS-APA checks whether the incoming set contains a fact $\widetilde{\gamma}$ such that $\langle\!\langle(\alpha_1, \ldots, \alpha_n, \texttt{f}), \varnothing\rangle\!\rangle \sqsupseteq \widetilde{\gamma}$. If this is the case, the analysis continues with a path edge to each successor t' of the statement `b=x.f` with a refined start fact: $\langle sp, \langle\!\langle(\alpha_1, \ldots, \alpha_n, \texttt{f}), \varnothing\rangle\!\rangle\rangle \rightarrow \langle t', \langle\!\langle \texttt{b}, \varnothing\rangle\!\rangle\rangle$. The intuition for this refinement is: Iff any access path starting with $(\alpha_1, \ldots, \alpha_n, \texttt{f})$ flows into statement sp, it will flow to an access path starting with `b` at statement t'. Note that IFDS-APA proceeds as just described only, iff `f` was not excluded, i.e., $\texttt{f} \notin E_\alpha$ and $\texttt{f} \notin E_\beta$.

[3]We omitted mapping the actual parameter to the formal parameter and the return value of the callee to the assigned value at the caller to keep the explanation brief. Mappings of local variables through call and return edges are actually performed as in the original IFDS framework.

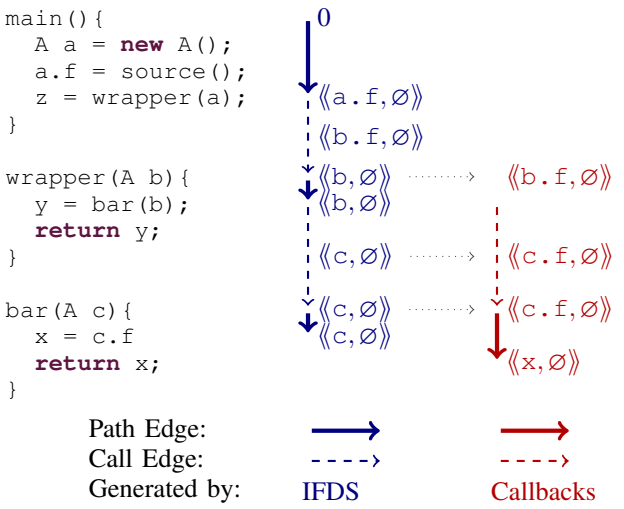

Fig. 3: Callbacks Created for a Field-Read Statement

If no caller requested the analysis with respect to `x.f` yet, the analysis must soundly cater for the fact that this could happen in later rounds of the fixed-point iteration. To handle this, IFDS-APA creates a callback that creates the path edge(s) upon future invocation. The callback is registered with the incoming set and is invoked if a caller provides an incoming fact `x.f`. Technically, we always create the callback, but this callback is invoked immediately if the condition is already fulfilled. Section IV-D discusses callbacks in more detail.

Field-Write Statements: Assume that we are analyzing a callee c with access paths represented by the bundle $\langle\!\langle x, \varnothing\rangle\!\rangle$, and c overwrites `x.f` with an un-tainted value, e.g., `x.f=null`. This assignment kills flows to `x.f`. Therefore, the analysis should only continue if a caller exists that calls c with an access path rooted in `x` but *different* from `x.f`. Field exclusions are used to express this. For a path edge $\langle sp, \langle\!\langle \alpha, E_\alpha\rangle\!\rangle\rangle \rightarrow \langle \texttt{x.f=null}, \langle\!\langle x, E_\beta\rangle\!\rangle\rangle$ the framework refines the start fact by adding the field `f` to the exclusion set, yielding the refined fact $\langle\!\langle \alpha, E_\alpha \cup \{\texttt{f}\}\rangle\!\rangle$. If the incoming set of c contains a fact $\widetilde{\gamma}$, such that $\langle\!\langle \alpha, E_\alpha \cup \{\texttt{f}\}\rangle\!\rangle \sqsupseteq \widetilde{\gamma}$, the analysis will continue with a path edge $\langle sp, \langle\!\langle \alpha, E_\alpha \cup \{\texttt{f}\}\rangle\!\rangle\rangle \rightarrow \langle t', \langle\!\langle x, E_\beta \cup \{\texttt{f}\}\rangle\!\rangle\rangle$ to each successor t' of statement `x.f=null`. If there is no incoming fact satisfying the constraint, the framework creates a callback that will create the path edge(s) on invocation. This is analogue to field reads.

D. Callbacks

Callbacks can be registered with abstraction points, e.g., the start point of a method together with trigger conditions that guard their invocation. A trigger condition specifies a predicate on an access-path bundle $\widetilde{\sigma}$. For example, the trigger condition for the callback registered with the method's starting point of the example in the sub-section about field-read statements, required an incoming fact $\widetilde{\gamma}$ for which $\langle\!\langle(\alpha_1, \ldots, \alpha_i, \texttt{f}), \varnothing\rangle\!\rangle \sqsupseteq \widetilde{\gamma}$ holds. In this case, the access-path bundle associated with the callback condition is $\widetilde{\sigma} = \langle\!\langle(\alpha_1, \ldots, \alpha_i, \texttt{f}), \varnothing\rangle\!\rangle$. The callback is then invoked if a fact $\widetilde{\gamma}$, for which $\widetilde{\sigma} \sqsupseteq \widetilde{\gamma}$ holds, is passed to the abstraction point with which the callback is registered.

Example 5: Consider the method `bar` in the example in

Figure 3, while ignoring the methods `main` and `wrapper` for now. The method `bar` will be analyzed with an access-path bundle $\langle\!\langle \mathtt{c}, \varnothing \rangle\!\rangle$ at the start point. Field `f` is read at statement `x=c.f`. This causes a callback cb to be registered with the start point $sp_{\mathtt{bar}}$ of `bar` that will create the path edge $\langle sp_{\mathtt{bar}}, \langle\!\langle \mathtt{c.f}, \varnothing \rangle\!\rangle\rangle \rightarrow \langle \mathtt{return\ x}, \langle\!\langle x, \varnothing \rangle\!\rangle\rangle$ upon invocation. The access-path bundle associated with cb as the trigger condition is $\widetilde{\sigma} = \langle\!\langle \mathtt{c.f}, \varnothing \rangle\!\rangle$. The incoming fact $\widetilde{\gamma} = \langle\!\langle \mathtt{c}, \varnothing \rangle\!\rangle$ is checked against this condition: $\widetilde{\sigma} \supseteq \widetilde{\gamma}$ does not hold – hence, no path edge is created. If the analysis processes another call that passes a taint $\widetilde{\alpha} = \langle\!\langle \mathtt{c.f}, \varnothing \rangle\!\rangle$ to the start point $sp_{\mathtt{bar}}$, the callback is triggered, because $\widetilde{\sigma} \supseteq \widetilde{\alpha}$ holds. The trigger condition also holds for $\widetilde{\alpha} = \langle\!\langle \mathtt{c.f.g}, \varnothing \rangle\!\rangle$, but not for $\widetilde{\alpha} = \langle\!\langle \mathtt{c.g}, \varnothing \rangle\!\rangle$.

The semantics of callback registration and invocation defined so far is simplistic. It ignores the fact that several abstractions can happen along different invocation chains: Callback trigger conditions so far only consider the local view on access paths of methods that contain field reads or writes. This may cause unsoundness of the analysis, as illustrated by the example below.

Example 6: Consider the methods `main` and `wrapper` in in Figure 3. The method `main` generates a taint $\langle\!\langle \mathtt{a.f}, \varnothing \rangle\!\rangle$ and passes it to `wrapper`. On `wrapper`'s entry, this taint is abstracted to $\langle\!\langle \mathtt{b}, \varnothing \rangle\!\rangle$ and then passed to `bar`, which reads the field `c.f`. This field read results in the callback installation with the trigger $\widetilde{\sigma} = \langle\!\langle \mathtt{c.f}, \varnothing \rangle\!\rangle$ as discussed before. The callback's condition cannot be fulfilled, as it is registered for the start point of `bar`, which was only passed the abstracted access path `c`, not `c.f`. As a result, the analysis will miss a tainted edge, hence be unsound.

The example not only illustrates how unsoundness can occur, if we leave things as described. It also suggests what needs to be done to avoid unsoundness: IFDS-APA must reiterate the registration and evaluation of callbacks in callers.

To ensure that callbacks are also triggered transitively, the framework *replicates* callbacks along the incoming sets. When the framework attaches a callback to a method start point, it iterates over the *incoming edges*, the path edges within the incoming set, and checks whether any of the edge's target facts *potentially satisfies* the condition of the callback. Assume an incoming edge $\langle sp_{caller}, \widetilde{\alpha} \rangle \rightarrow \langle t, \widetilde{\gamma} \rangle$ for a callee start point sp_{callee}. The incoming fact $\widetilde{\gamma}$ *potentially satisfies* the callback condition $\widetilde{\sigma}$ in two cases: (a) if $\widetilde{\gamma} \supseteq \widetilde{\sigma}$ or (b) if $\widetilde{\gamma} \approx \widetilde{\sigma}$. In any of the two cases, the framework performs the following steps: (a) it computes the delta δ between the incoming fact $\widetilde{\gamma}$ and the condition $\widetilde{\sigma}$, $\delta = \Delta(\widetilde{\gamma}, \widetilde{\sigma})$; (b) the delta is concatenated to the start fact $\widetilde{\alpha}$ of the incoming edge; (c) the result $\widetilde{\sigma'} = \widetilde{\alpha} :: \delta$ is installed as the trigger condition for the replicated callback, and (d) the latter is registered with the start point sp_{caller} of the incoming edge. On invocation of the replicated callback, the framework registers a new incoming fact to the start point sp_{callee}, constructed by concatenating the original incoming $\widetilde{\gamma}$ fact and the computed delta δ. This new incoming fact satisfies the trigger condition of the original callback at sp_{callee}, which therefore will be executed.

Note that when the framework registers the replicated callback with the start point of the incoming edge, this may result in other replicated callbacks being created as the same steps are performed recursively for callers of callers. This recursion may happen until the analysis reaches an initial seed (**0**-Fact). This process may seem expensive at first sight, but it is not. Consider that the framework only needs to traverse path edges. Each path edge skips over all the internal nodes of the respective methods, such that in the worst case one needs to traverse only as many path edges as there are frames on the current abstract call stack.

The following example illustrates how the transitive registering of callbacks avoids the unsoundness that would occur otherwise.

Example 7: Reconsider Example 6 and Figure 3. The field-read statement in method `bar` will generate a callback. This callback creates the path edge $\langle sp_{\mathtt{bar}}, \langle\!\langle c.f, \varnothing \rangle\!\rangle\rangle \rightarrow \langle \mathtt{return\ x}, \langle\!\langle x, \varnothing \rangle\!\rangle\rangle$. The trigger condition $\widetilde{\sigma}$ of the callback is not satisfied by the incoming fact $\langle\!\langle c, \varnothing \rangle\!\rangle$, because $\widetilde{\sigma} = \langle\!\langle c.f, \varnothing \rangle\!\rangle \not\supseteq \langle\!\langle c, \varnothing \rangle\!\rangle$. But, this incoming fact potentially satisfies the condition transitively, because $\langle\!\langle c, \varnothing \rangle\!\rangle \supseteq \langle\!\langle c.f, \varnothing \rangle\!\rangle$. Consequently, the framework replicates a transitive callback for the incoming edge, reflecting the call edge from `wrapper` to `bar`. Using the path edge into the call site, the framework can retrieve the start-point statement and the start fact at that statement. The transitive callback is registered with this start-point statement using as condition the start fact concatenated with the delta $\Delta(\langle\!\langle c, \varnothing \rangle\!\rangle, \langle\!\langle c.f, \varnothing \rangle\!\rangle)$. Therefore, the condition is $\langle\!\langle b, \varnothing \rangle\!\rangle :: \Delta(\langle\!\langle c, \varnothing \rangle\!\rangle, \langle\!\langle c.f, \varnothing \rangle\!\rangle) = \langle\!\langle b.f, \varnothing \rangle\!\rangle$. This condition is immediately satisfied by the incoming fact $\langle\!\langle b.f, \varnothing \rangle\!\rangle$ that is passed from `main` to `wrapper`. Hence, the transitive callback is invoked immediately, and creates the call edge, which registers $\langle\!\langle c.f, \varnothing \rangle\!\rangle$ as incoming fact to `bar`. This new incoming fact triggers the first callback, which in turn creates the path edge $\langle sp_{\mathtt{bar}}, \langle\!\langle c.f, \varnothing \rangle\!\rangle\rangle \rightarrow \langle \mathtt{return\ x}, \langle\!\langle x, \varnothing \rangle\!\rangle\rangle$, with which the analysis will continue.

E. Loops

To solve the termination problem caused by recursive constructs such as loops and recursive data structures *without* bounding the size of access paths, Access-Path Abstraction introduces additional abstraction points beyond method calls. Here we explain the handling of other control-flow merge points. This includes, in particular, entries to loops. First, we illustrate by an example, how the termination problem is solved by introducing loops as abstraction points. Subsequently, we discuss the algorithmic changes required to accommodate additional abstraction points.

Example 8: For the loop shown in Figure 2b the framework will proceed as follows. It first creates an initial self-loop at the start point of `foo` with the access-path bundle $\langle\!\langle \mathtt{a}, \varnothing \rangle\!\rangle$. Then it passes this bundle to the beginning of the loop, i.e., the abstraction point. The framework abstracts the bundle, which in this case has no effect, yielding the same bundle again. The loop causes the bundle $\langle\!\langle \mathtt{b.f}, \varnothing \rangle\!\rangle$ to be created along one branch and $\langle\!\langle \mathtt{b.g}, \varnothing \rangle\!\rangle$ along the other. Then, `b` is assigned to `a`, such that $\langle\!\langle \mathtt{a.f}, \varnothing \rangle\!\rangle$ and $\langle\!\langle \mathtt{a.g}, \varnothing \rangle\!\rangle$ are passed back to the entry of the loop, i.e., back to the abstraction point. At the loop abstraction point, the framework proceeds just as it does at the entry to a called procedure. It stores both bundles as incoming facts and continues the next iteration with abstracted access-path bundles. In this example, no additional iteration will be computed, because both access-path bundles are abstracted to $\langle\!\langle \mathtt{a}, \varnothing \rangle\!\rangle$, i.e., to a fact that the analysis already processed.

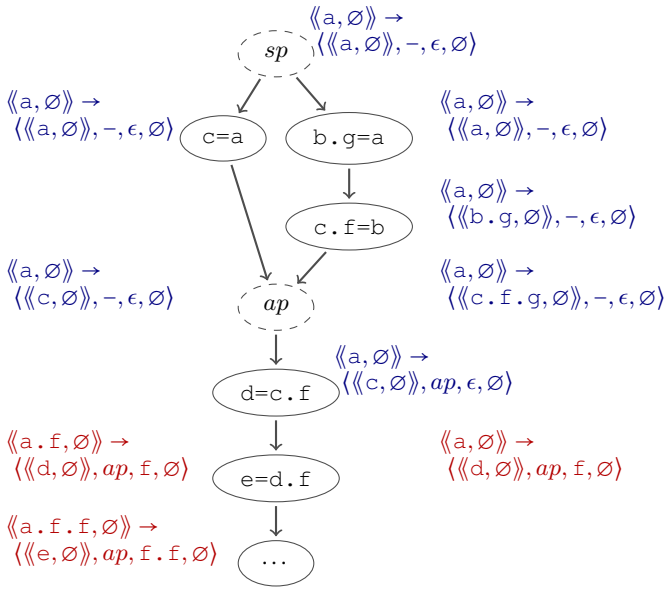

Fig. 4: Handling of Arbitrary Nodes as Abstraction Points
$\langle\langle\alpha,\varnothing\rangle\rangle \rightarrow \langle\langle\beta,\varnothing\rangle, ap, \gamma, \varnothing\rangle$ is used here as abbreviation for a path edge $\langle sp, \langle\langle\alpha,\varnothing\rangle\rangle \rightarrow \langle s, \langle\langle\beta,\varnothing\rangle\rangle, ap, \gamma, \varnothing\rangle$, whereby s is the respective CFG node at which the path edge is drawn.

To summarize, the additional abstraction points solve the termination problem without the need to restrict the size of access paths. However, the additional abstraction points require an algorithmic change to correctly handle callbacks. When a field-read or field-write statement is processed, the framework must register a callback with the latest abstraction point it passed. If that abstraction point is a method entry, it is easily identified, because every path edge starts at a method start point by definition.

To identify the latest abstraction point in other situations, we modify the definition of path edges to also use a reference to the latest non-start-point abstraction point if such an abstraction point has been passed. In addition, we include a sequence of fields, to reflect which field accesses have been resolved via the abstraction point already, and a set of exclusions. We will explain their need in the next example. From now on, we will write path edges as $\langle sp, \widetilde{\alpha}\rangle \rightarrow \langle t, \widetilde{\beta}, ap, \gamma, E_\gamma\rangle$, whereby $sp, \widetilde{\alpha}, t$, and $\widetilde{\beta}$ are—as before—the start point, access-path bundle at the start point, a target statement, and an access-path bundle at the target, respectively. New are ap, γ, and E_γ representing an abstraction point, a sequence of field accesses $\gamma = (\gamma_1, \ldots, \gamma_n), \gamma_i \in \mathcal{F}$, and a set of excluded fields $E_\gamma \subset \mathcal{F}$.

Example 9: For this example, consider Figure 4. The figure shows a control-flow graph supplemented by artificial nodes for the start point sp and an abstraction point ap. Each node is annotated with the path edge targeting that node, whereby we use an abbreviated form that omits the statements, as they are easily inferred from the graph. We show only those path edges that are of interest for the example. Note that path edges are aligned at the left side of the figure if a path edge results from the left branch taken, aligned at the right side if the right branch is taken, respectively. The path edge is drawn close to the node, if the path edge results from both branches.

The path edges in the upper part of the figure, drawn in blue, are straightforward. The analysis propagates through both branches resulting in facts $\langle\langle \texttt{c},\varnothing\rangle\rangle$ and $\langle\langle \texttt{c.f.g},\varnothing\rangle\rangle$ passed to the abstraction point ap. Both are registered as incoming facts, together with their respective path edges. After the abstraction point, the analysis continues with an abstracted access-path bundle $\langle\langle \texttt{c},\varnothing\rangle\rangle$ only, and stores in the path edge succeeding ap a reference to ap. Now, at statement `d=c.f` the field `f` is read, resulting in a callback being generated. The callback is not registered with the start point sp, but with the abstraction point ap, because in the path edge an abstraction point was referenced, i.e., the start point is not the latest passed abstraction point. The callback's condition is checked against the incoming facts at ap. The condition can be immediately fulfilled via the right branch providing $\langle\langle \texttt{c.f.g},\varnothing\rangle\rangle$. This yields the path edge $\langle sp, \langle\langle \texttt{a},\varnothing\rangle\rangle\rangle \rightarrow \langle \texttt{e=d.f}, \langle\langle \texttt{d},\varnothing\rangle\rangle, ap, \texttt{f}, \varnothing\rangle$.

Note that the framework includes the field `f` in the path edge, reflecting that a condition was fulfilled consuming a field access `f` of some incoming edge at ap. This is because the analysis satisfied the callback's condition via the incoming fact $\langle\langle \texttt{c.f.g},\varnothing\rangle\rangle$ "consuming `f`" such that for succeeding field-read statements the analysis must take into account that `f` has been read already and the incoming fact can now only be used to read field `g`. Proceeding in the example, IFDS-APA uses this information to know at statement `e=d.f` that the callback generated here cannot be fulfilled by ap via the right branch. To store information about which fields have already been excluded, the set E_γ of the new path edge is used analogously.

So far, we have ignored that the callback generated at statement `d=c.f` could be potentially satisfied transitively through the start point sp. This is the case, because the incoming fact via the left branch potentially satisfies the condition. As discussed in Section IV-D, IFDS-APA therefore replicates the callback for transitively preceding abstraction points, i.e., for the start point sp. When this replicated callback can be fulfilled, which in this case depends on the incoming facts provided by callers, IFDS-APA registers an incoming fact $\langle\langle \texttt{c.f},\varnothing\rangle\rangle$ at ap, which triggers the creation of path edge $\langle sp, \langle\langle \texttt{a.f},\varnothing\rangle\rangle\rangle \rightarrow \langle \texttt{e=d.f}, \langle\langle \texttt{d},\varnothing\rangle\rangle, ap, \texttt{f}, \varnothing\rangle$.

F. Return Sites

By introducing loops as additional abstraction points, we solve the state explosion problem for the intra-procedural case. But, state explosion may still occur inter-procedurally.

For illustration, consider the example in Figure 5. Assume `Foo` to be an interface declaring a method `foo` with a parameter `x` to be tainted. Classes `A` and `B` both implement this interface and provide similar implementations of `foo`: Both recursively call `foo`. Assume that the precise type of `f` is statically undecidable, therefore, the call is resolved to `A.foo` and `B.foo`. In addition, both wrap the tainted value in another object's field and return the object: `A` writes to field `a` and `B` to field `b`. Assume the analysis is computing a summary for a tainted parameter `x`. Without the recursive call, the summary of `foo` yields a mapping from `x` to `y.a`, respectively `y.b`. For the branch performing the recursive call, the analysis applies the summaries for both `A.foo` and `B.foo`, which will in turn generate the additional summaries from `x` to `y.a.a`, and `y.a.b` for `A.foo` and from `x` to `y.b.a`, and `y.b.b` for `B.foo`. This results in a similar state explosion as for loops.

```
class A impl. Foo {
  Foo f = aOrB();
  foo(X x) {
    if(unknown())
      x = f.foo(x);
    y.a = x;
    return y;
  }
}
```

```
class B impl. Foo {
  Foo f = aOrB();
  foo(X x) {
    if(unknown())
      x = f.foo(x);
    y.b = x;
    return y;
  }
}
```

Fig. 5: State Explosion through Access Paths

Note that an analog case, in which the taint is wrapped in a field before passing it as parameter to the recursive call, is already solved. As IFDS-APA abstracts at method start points, the analysis encounters the previously seen fact $\langle\!\langle \mathtt{x}, \varnothing \rangle\!\rangle$ and stops. By also applying abstraction at return sites, in the same way as in the treatments for loops, the state explosion problem can be solved here as well. Hence, at the return sites of call `foo` the framework does abstract that the returned facts are $\langle\!\langle \mathtt{x.a}, \varnothing \rangle\!\rangle$ and $\langle\!\langle \mathtt{x.b}, \varnothing \rangle\!\rangle$ and continues with the fact $\langle\!\langle \mathtt{x}, \varnothing \rangle\!\rangle$. $\langle\!\langle \mathtt{x.a}, \varnothing \rangle\!\rangle$ and $\langle\!\langle \mathtt{x.b}, \varnothing \rangle\!\rangle$ are registered as incoming facts with the return site. As before, a reference to the return site as last passed abstraction point is included in subsequent path edges allowing to reconstruct the abstracted field accesses.

G. Termination in Presence of an Unbounded Domain

In theory, the analysis domain consisting of access-path bundles has an infinite size. One thus might wonder why IFDS-APA guarantees termination. Termination would be threatened in cases where the framework would create access-path bundles of ever-growing size. But this is impossible, as such unlimited growth can only occur due to loops or recursion. Since IFDS-APA abstracts access-path bundles at every entry into a procedure or loop, the length of the access-path portion of an access-path bundle is thus bounded by the maximal length of the access paths used within a single procedure or loop iteration. This guarantees termination.

V. Evaluation

We performed experiments to compare the proposed approach against two baseline approaches: A field-based approach, denoted FB, and a classic field-sensitive approach that uses k-limiting, as described in Section III, denoted FS_k. Specifically, the experiments address the following two research questions:

- RQ1: Given a fixed heap size, which analyses can successfully analyze our benchmark subjects?
- RQ2: How fast is IFDS-APA compared to the baseline approaches?

A. Setup

Our implementation of IFDS-APA is based on Heros [6], an open-source implementation of an IFDS/IDE solver; we contributed our adaptations back to the Heros project. The two baseline approaches also use the IFDS implementation provided by Heros. The experiments carry out taint analyses, for which we use the implementation of FlowTwist [8]. FlowTwist uses Heros and is based on the Soot code-analysis framework [9]. Originally designed to address confused-deputy problems in the Java Class Library, FlowTwist can be used to conduct general-purpose data-flow analysis [10].

We use FlowTwist for three different experimental setups. In the first two setups we use an adaptation of FlowTwist to detect SQL injection, command injection, path traversal, and unchecked redirection vulnerabilities. We apply the analysis to the Stanford SecuriBench [11] dataset consisting of seven web applications. In the first setup we use only the bare web applications, while we include their dependencies (and the Java Class Library) in the second setup. For the first two setups a pure forward analysis is conducted. In a third setup we use the original FlowTwist implementation, which conducts a synchronized forward and backward taint-analysis to detect confused-deputy vulnerabilities within the Java Class Library (JCL) 1.7.0, e.g., any call to the method `Class.forName(String cls)`, where (1) the String `cls` is user-controlled, and (2) the return value flows back to the user. These flows are problematic and are commonly used in exploits [8]. This setup uses a call graph starting at all of the JCL's public methods, leading to a much larger coverage of the JCL's methods than with SecuriBench.

The applications within the SecuriBench suite vary from 32 to 445 classes and 4,191 to 52,089 lines of code per project. We found it much more relevant, though, to characterize the projects by the number of edges of their respective inter-procedural control flow graphs (ICFGs), which are shown in the second column of Table I. The ICFGs are relatively small if the web applications are considered in isolation, but their size grows significantly if all dependencies are also considered. We counted only those control-flow edges that are contained in methods that are transitively reachable from within the web applications.

All experiments were conducted on a machine running OS X 10.10 with a 4-core Intel Xeon E5 3.0 GHz processor and 32 GB memory. As Java Runtime Environment, we used the Oracle Java 1.8.0u40 release, with a heap size set to a maximum of 25 GB.

B. Results

RQ1 seeks to answer the questions which approaches can at all analyze which benchmarks within the allotted 25 GB of maximum heap size. To address this question, we ran all approaches on all benchmarks. Table II shows those configurations that ran out of memory as OoM. Results for all three setups are shown separated by horizontal lines. As long as the dependencies were excluded, all approaches were able to analyze all of SecuriBench. If dependencies are considered, however, only IFDS-APA and FB were able to analyze all SecuriBench applications. FS_k was only able to analyze all of SecuriBench when k is set to zero. *blueblog* and *webgoat* could be analyzed for k-limiting with $\mathrm{FS}_{k=1}$ or $\mathrm{FS}_{k=2}$, but no application could be analyzed with any higher value for k. None of the configurations was able to successfully complete the FlowTwist analysis of Java 1.7.0 with the available memory.

To gain a better understanding of the relative scalability, we measured two metrics while running all subject analyses on the web applications of SecuriBench. First, we measured the fraction of the statements each analysis must traverse, i.e., how many ICFG-edges it traverses. We also measured how many times flow functions are being evaluated. The results of these measurements are shown in Table I. The fraction of

TABLE I: Measures of Efforts Spent by each Analyses
Vis.: ICFG edges visited during the analysis; Eval.: Number of flow-function evaluations; OoM: Out of memory

	Project	ICFG Edges	IFDS-APA		FB		FS_k $k=0$		$k=1$		$k=2$		$k=3$	
			Vis.	Eval.	Vis.	Eval.	Vis.	Eval.	Vis.	Eval.	Vis.	Eval.	Vis.	Eval.
Excl. Dependencies	blueblog	8 529	11%	2 652	29%	10 438	33%	10 441	32%	20 133	33%	30 981	33%	39 823
	jboard	14 154	2%	524	1%	660	2%	614	2%	614	2%	614	2%	614
	pebble	67 488	26%	61 676	25%	74 395	27%	66 626	26%	73 126	26%	73 643	26%	74 104
	personalblog	11 391	13%	8 956	14%	11 376	14%	7 036	14%	7 024	14%	7 024	14%	7 024
	roller	82 264	2%	4 069	3%	9 156	3%	7 942	2%	6 658	2%	6 658	2%	6 658
	snipsnap	137 532	6%	22 799	7%	29 234	10%	35 571	7%	23 651	7%	25 500	7%	26 249
	webgoat	15 122	26%	13 694	22%	12 927	27%	13 526	27%	13 083	27%	13 083	27%	13 083
Incl. Dependencies	blueblog	692 483	3%	101 302	8%	174 713	46%	3 460 269	29%	5 751 330	29%	7 131 355	OoM	OoM
	jboard	2 353 761	14%	1 610 077	30%	6 287 863	61%	21 644 040	OoM	OoM	OoM	OoM	OoM	OoM
	pebble	1 769 459	13%	1 112 081	27%	3 237 303	62%	17 128 299	OoM	OoM	OoM	OoM	OoM	OoM
	personalblog	2 194 345	15%	1 664 999	28%	4 839 828	62%	27 837 129	OoM	OoM	OoM	OoM	OoM	OoM
	roller	2 891 553	15%	2 286 229	27%	6 999 622	56%	15 536 275	OoM	OoM	OoM	OoM	OoM	OoM
	snipsnap	2 683 739	14%	1 935 187	28%	17 207 797	63%	26 382 020	OoM	OoM	OoM	OoM	OoM	OoM
	webgoat	734 345	16%	692 080	20%	1 349 476	52%	4 612 198	41%	11 615 079	44%	29 299 712	OoM	OoM
	JCL 1.7.0	12 069 342	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM

TABLE II: Run Times of the IFDS Framework in Seconds

	Project	IFDS-APA	FB	FS_k $k=0$	$k=1$	$k=2$	$k=3$
Excl. Dependencies	blueblog	0.22	0.32	0.44	0.29	0.26	0.34
	jboard	0.02	0.02	0.04	0.01	0.01	0.01
	pebble	0.99	0.70	0.81	0.53	0.48	0.49
	personalblog	0.14	0.07	0.07	0.05	0.05	0.05
	roller	0.07	0.06	0.07	0.05	0.05	0.05
	snipsnap	0.32	0.24	0.35	0.21	0.26	0.22
	webgoat	0.16	0.10	0.15	0.10	0.10	0.10
Incl. Dependencies	blueblog	1.21	1.05	27.15	43.54	54.56	OoM
	jboard	322.70	40.81	228.84	OoM	OoM	OoM
	pebble	108.38	17.13	138.40	OoM	OoM	OoM
	personalblog	202.08	24.92	236.65	OoM	OoM	OoM
	roller	478.81	35.19	102.83	OoM	OoM	OoM
	snipsnap	307.65	113.01	203.16	OoM	OoM	OoM
	webgoat	57.75	6.70	30.86	98.14	253.56	OoM
	JCL 1.7.0	OoM	OoM	OoM	OoM	OoM	OoM

ICFG-edges each analysis traverses is shown as a percentage of all the application's ICFG edges in columns denoted as *Vis*. The frequency of evaluating flow functions is shown as an absolute number in columns denoted as *Eval*. While the percentage of visited ICFG edges is similar for all approaches when analyzing the web applications in isolation, we can clearly see a trend when dependencies are considered: IFDS-APA visits fewer ICFG edges than the other approaches. The same trend is visible for the number of flow-function evaluations. For FS_k, with increasing k the analysis must compute more flow functions (for more contexts) but can sometimes restrict itself to a slightly smaller fraction of the ICFG, due to the added precision.

To address the second research question, we measured the execution time of the IFDS framework, excluding the time taken to load and pre-process the bytecode as well as to compute a call graph. This pre-computation time is shared by all three approaches. The results are shown in Table II. All approaches are able to analyze the web applications in isolation in less than a second. When dependencies are considered, we can observe differences in execution times ranging from one second up to 8 minutes for a single approach, depending on the benchmark. We can see that the FB is faster than IFDS-APA. $FS_{k=0}$ is faster than IFDS-APA on some applications and slower on others. For $k > 0$, FS_k is slower than IFDS-APA.

C. Discussion

Regarding RQ1 we conclude that memory wise IFDS-APA scales clearly better than FS_k and as well as FB. FS_k fails for higher k values when analyzing SecuriBench with dependencies. All approaches are not able to terminate successfully within the given memory when analyzing the Java Class Library.

The answer to RQ2 is not as clear. Especially when including dependencies IFDS-APA is faster than FS_k for some benchmarks but slower for others. It is also worth noting that IFDS-APA is always slower than FB, *although* FB typically computes more flow functions and traverses larger portions of the ICFG due to its rather imprecise abstraction that leads to over-tainting. However, while FB can traverse the entire application in a single fixed-point iteration, IFDS-APA must reconstruct abstracted access-paths on demand and must proactively register callbacks to allow for incoming access paths computed in later phases of the fixed-point iteration. Our experiments indicate that the cost of these operations seem to outweigh savings due to the more precise abstraction.

Before concluding this section, we briefly consider precision. By nature, a field-sensitive approach can yield more precise results than a field-based approach (see Figure 1b). In k-limiting, over-approximation is controlled by the value for k. In theory, k-limiting would achieve the same precision as IFDS-APA, if k is chosen to be at least the length of the longest access path in the application to be analyzed. This is, considering an application which does not generate access paths of infinite length through loops or recursion as shown in Example 3. Note that, in contrast, IFDS-APA terminates and achieves optimal precision (with respect to field sensitivity) even for such programs.

As our experiments further show, choosing a high value for k will severly degrade scalability of a k-limiting based approach in practice. On the other hand, selecting small k-values not only degrades precision, but may as well be a threat to the scalability, as more data flows have to be considered that are caused by the over-approximation. This is indicated by the results shown in Table I. For example, the application *snipsnap* excluding dependencies has more flow-function evaluations for $FS_{k=0}$ than for $FS_{k=1}$. In addition, the number of flow-function evaluations increases again for k values set to two or three. Further, a too

small k-value can even give up soundness. Using IFDS-APA relieves analysis designers of all those considerations.

We conclude that IFDS-APA implements an analysis that in terms of precision is at least as precise as FS_k and scales better than FS_k for all values of $k \geq 0$.

VI. Related Work

Despite the existence of many data-flow frameworks, we are not aware of any other framework explicitly handling field-sensitivity and bounding the data-flow domain. Both is usually left to the clients of the framework. Therefore, we here relate to *field-sensitive data-flow analyses* and how they model data-flow domains, and to existing work on *abstract summaries*.

A. Field-Sensitive Data-Flow Models

The access-path model is broadly used within analyses, such as alias analyses [3] or taint analyses [1], [2].

One attempt by Deutsch [12] to circumvent the limit of the access-path model was to use a *symbolic representation of an access path* in which reoccurring field accesses are grouped into a single symbolic one. The symbolic notation is close to a regular expression over the fields. For example, if two aliased values are both repeatedly written to a field `f` in the same loop, Deutsch's approach is able to learn that $\texttt{a.f}^n$ and $\texttt{b.f}^n$ may be aliased, whereas n is some arbitrary number of times the aliased values are nested. The advantage of the approach is that it is known that the nesting has happened the same times for both values and that only the n-th nesting is aliased with each other. While this is a solution to overcome k-limiting in this special case, it does not solve the general case. If only one value is considered, n has no more meaning. This results in a simple over-approximation comparable to a variant of k-limiting as it is applied in FlowDroid.

FlowDroid [2] is a taint analysis for Android applications. In addition to limiting the access path to be at most of size k, FlowDroid collapses sub-paths between two equal field accesses in an access path. If a sub-path is collapsed, FlowDroid flags that this sub path may be repeatedly read. This is an over-approximation and may result in fields being read for which a taint has never been written.

Geffken et al. [13] propose an inter-procedural side-effect analysis. To ensure field-sensitivity, they extend Deutsch's symbolic access path to a *generalized access graph*, which models field accesses as a directed graph; reappearing field accesses by the same statement correspond to cycles in the graph. This ensures termination without requiring over-approximations like k-limiting. So far, they only tested their analysis on a small benchmark. They do not completely solve the state explosion problem shown in Figure 5, which is why we expect scalability issues on benchmarks that include similar program constructs.

In alias analyses it is also common [14]–[16] to express the alias relation within a *context-free language* (CFL) and therefore solve a reachability problem over that language. Fields are part of the language and recursive field accesses are grouped into arbitrary accesses using a wildcard leading to an over-approximation.

B. Abstract Summaries

In [17] Chandra et al. introduce a technique of *generalization* to produce summaries which are applicable to many data-flow facts. As the proposed tool Snugglebug reasons about weakest preconditions along the control flow to reach a certain statement, their data-flow domain consists of conditions. Hence, their generalization technique differs from ours.

The framework proposed by Yorsh et al. [18] is a more theoretical approach on how to gain more concise summaries by composing the flow-functions and their preconditions. As examples they conduct a typestate analysis and constant propagation. Within their typestate analysis they reason about fields by using 1-limiting, within constant propagation they do not handle fields.

Landi and Ryder [19] used in their alias analysis an approach for which they abstracted access paths as *non-visible* inside callees. Using this technique the analysis results for the procedure become reusable across multiple calling contexts. When evaluating returns they restore access paths according to the respective calling contexts. Our approach was independently developed, but the abstraction at method start points is very similar to their approach. Yet, the previous work lacks general support for arbitrary abstraction points, which is why it still requires over-approximations to limit the size of access paths and does not address the state-explosion problem.

Jensen et al. [20] represent in their abstraction the whole state of the heap. As they point out, this makes summaries nearly impossible to be reused. To obtain more reusable summaries, they therefore represent properties of the heap as *unknown* and recover properties as soon as they are accessed. They call this concept lazy propagation as properties are propagated into callees on-demand. When applying summaries they replace unknown properties by the values available in the calling context. The idea of abstracting at calls and recovering abstracted state is very similar to ours. Yet, we are the first to show that if applying it at loops and return edges as well one can remove the need of over-approximations through k-limiting, thus solving the state-explosion problem.

VII. Conclusion

Within this work we have presented Access-Path Abstraction, our extension to the IFDS framework to support field-sensitivity within an arbitrary data-flow analysis. Without much effort, any analysis using IFDS-APA can easily yet precisely reason about fields. As our experiments have shown the analysis will largely gain in terms of efficiency through the precise and abstract summaries IFDS-APA uses internally: For the first time, we managed to scale the tough FlowTwist analysis on the whole Java Class Library with field sensitivity – so far even the field-based approach did not scale. Finally, introducing the more abstract summaries, we also managed to abandon any need for k-limiting within the access-path model, herein we see a huge profit for existing and future analyses.

Acknowledgments

This work was supported by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, by the DFG Collaborative Research Center CROSSING and the Emmy Noether Group RUNSECURE.

REFERENCES

[1] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, 2013, pp. 210–225.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "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*, ser. PLDI '14, 2014, pp. 259–269.

[3] A. De and D. D'Souza, "Scalable flow-sensitive pointer analysis for java with strong updates," in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ser. ECOOP'12, 2012, pp. 665–687.

[4] N. D. Jones and S. Muchnick, "Flow analysis and optimization of lisp-like structures," in *In Proceedings of the Symposium on Principles of Programming Languages (POPL)*, ser. POPL '79, 1979, pp. 244–256.

[5] 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*, ser. POPL '95, 1995, pp. 49–61.

[6] 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*, ser. SOAP '12, 2012, pp. 3–8.

[7] N. A. Naeem and O. Lhotak, "Typestate-like analysis of multiple interacting objects," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, ser. OOPSLA '08, 2008, pp. 347–366.

[8] J. Lerch, B. Hermann, E. Bodden, and M. Mezini, "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*, ser. FSE '14, 2014, pp. 98–108.

[9] 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*, ser. CETUS '11, 2011.

[10] J. Lerch and B. Hermann, "Design your analysis: A case study on implementation reusability of data-flow functions," in *4th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, ser. SOAP '15, 2015.

[11] B. Livshits, "Stanford SecuriBench," http://suif.stanford.edu/~livshits/securibench/, 2005, version 91a.

[12] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94, 1994, pp. 230–241.

[13] M. Geffken, H. Saffrich, and P. Thiemann, "Precise interprocedural side-effect analysis," in *Theoretical Aspects of Computing*, ser. ICTAC '14, 2014, pp. 188–205.

[14] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven points-to analysis for java," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '15, 2005, pp. 59–76.

[15] D. Yan, G. H. Xu, and A. Rountev, "Demand-driven context-sensitive alias analysis for java," in *Proceedings of the 20th International Symposium on Software Testing and Analysis*, ser. ISSTA '11, 2011, pp. 155–165.

[16] G. H. Xu, A. Rountev, and M. Sridharan, "Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis," in *Proceedings of the 23rd The European Conference on Object-Oriented Programming*, ser. ECOOP '09, 2009, pp. 98–122.

[17] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: a powerful approach to weakest preconditions," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, 2009, pp. 363–374.

[18] G. Yorsh, E. Yahav, and S. Chandra, "Generating precise and concise procedure summaries," in *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '08, 2008, pp. 221–234.

[19] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural aliasing," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, ser. PLDI '92, 1992, pp. 235–248.

[20] S. H. Jensen, A. Møller, and P. Thiemann, "Interprocedural analysis with lazy propagation," in *Proceedings of the 17th International Conference on Static Analysis*, ser. SAS'10, 2010, pp. 320–339.