IDE^{al}: Efficient and Precise Alias-Aware Dataflow Analysis

JOHANNES SPÄTH, Fraunhofer IEM, Germany
KARIM ALI, University of Alberta, Canada
ERIC BODDEN, Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM, Germany

Program analyses frequently track objects throughout a program, which requires reasoning about aliases. Most dataflow analysis frameworks, however, delegate the task of handling aliases to the analysis clients, which causes a number of problems. For instance, custom-made extensions for alias analysis are complex and cannot easily be reused. On the other hand, due to the complex interfaces involved, off-the-shelf alias analyses are hard to integrate precisely into clients. Lastly, for precision many clients require strong updates, and alias abstractions supporting strong updates are often relatively inefficient.

In this paper, we present IDE^{al}, an alias-aware extension to the framework for Interprocedural Distributive Environment (IDE) problems. IDE^{al} relieves static-analysis authors completely of the burden of handling aliases by automatically resolving alias queries on-demand, both efficiently and precisely. IDE^{al} supports a highly precise analysis using strong updates by resorting to an on-demand, flow-sensitive, and context-sensitive all-alias analysis. Yet, it achieves previously unseen efficiency by propagating aliases individually, creating highly reusable per-pointer summaries.

We empirically evaluate IDE^{al} by comparing TS^{f}, a state-of-the-art typestate analysis, to TS^{al}, an IDE^{al}-based typestate analysis. Our experiments show that the individual propagation of aliases within IDE^{al} enables TS^{al} to propagate 10.4× fewer dataflow facts and analyze 10.3× fewer methods when compared to TS^{f}. On the DaCapo benchmark suite, TS^{al} is able to efficiently compute precise results.

CCS Concepts: • **Software and its engineering → Software defect analysis**;

Additional Key Words and Phrases: static analysis, dataflow, aliasing

ACM Reference Format:
Johannes Späth, Karim Ali, and Eric Bodden. 2017. IDE^{al}: Efficient and Precise Alias-Aware Dataflow Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 99 (October 2017), 27 pages. https://doi.org/10.1145/3133923

1 INTRODUCTION

Tracking object states helps static analyses derive information about the quality and security of a given software. For example, typestate analysis [Alur et al. 2005; Fink et al. 2008; Naeem and Lhoták 2008; Udrea and Lumezanu 2006; Whaley et al. 2002; Yahav and Ramalingam 2004] detects programming errors that drive an object into an undesirable state. Shape analysis [Ghiya and Hendren 1996; Sagiv et al. 1999] proves program and data-structure invariants, particularly by reasoning about links between objects that are allocated on the heap. This information is useful for finding software bugs such as memory leaks [Dor et al. 2000] or detecting security vulnerabilities by tracking the propagation of taints (i.e., private data) [Arzt et al. 2014].

Authors' addresses: Johannes Späth, Fraunhofer IEM, Germany, johannes.spaeth@iem.fraunhofer.de; Karim Ali, University of Alberta, Canada, karim.ali@ualberta.ca; Eric Bodden, Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM, Germany, eric.bodden@uni-paderborn.de.

2475-1421/2017/10-ART99
https://doi.org/10.1145/3133923

```
1 File a = new File();
2 b = a;
3 b.open();
4 a.close();
```

Fig. 1. An example that depicts the importance of handling aliases in a static dataflow analysis.

Programs can access and manipulate instantiated objects through different pointers, some of which may not be visible in the current analysis scope. To compute correct results, a static analysis must therefore properly handle *aliasing*. For example, the code snippet in Figure 1 allocates a `File` object (line 1) and assigns it to the variable `a`. It then copies `a` to `b` (line 2), causing `a` and `b` to alias. The code then calls `open` on `b` (line 3) and `close` on `a` (line 4). If a typestate analysis that checks for unclosed files does not properly handle aliases, it will imprecisely report that `b` might not be closed by the end of the program. A *precise* typestate analysis should instead report that `b` is closed as soon as `close` is called on `a`, because `b` and `a` are aliases to the same `File` object. On the other hand, aliasing is also a requirement for *sound* static analyses. That is, the analysis shall not miss dataflows that may occur at runtime. A key challenge of static analysis is to design analyses that are both precise and sound at the same time, and, ideally, implemented in an efficient manner. We identify aliasing as a major component of the challenge and recognized a number of key deficiencies in the way that most static analyses currently handle aliasing.

Deficiency 1: Custom-made extensions for alias analysis are complex and cannot easily be reused. The implementation of static analyses are greatly simplified by the use of dataflow analysis frameworks. However, using such frameworks does not simplify the handling of aliasing: Virtually all existing dataflow analysis frameworks [Padhye and Khedker 2013; Reps et al. 1995; Sagiv et al. 1996] offload the burden of handling aliases to the client analysis, e.g., typestate, shape, or taint analysis, forcing the designers of those static-analysis clients to consider, design, and implement their own aliasing solutions. A common and intuitive solution is to embed the alias relationships into the client analysis' dataflow functions. However, this solution makes the alias analysis heavily intertwined with the client analysis, preventing it from being reused for other clients.

Deficiency 2: Off-the-shelf alias analyses are hard to integrate well into client analyses. Because of the problems mentioned above, static-analysis designers instead often resort to the use of an off-the-shelf alias analysis. This option comes with its own set of problems though. In particular, it is highly non-trivial to integrate an off-the-shelf analysis in such a way that the resulting analysis as a whole is maximally precise and efficient [Späth et al. 2016]. For optimal precision and performance, client and alias analyses should integrate in a flow-sensitive, context-sensitive, and demand-driven manner such that aliases are only computed when necessary and only for calling contexts and program locations in which they matter to the client analysis.

Deficiency 3: Current precise alias abstractions, if supporting strong updates, are relatively inefficient. When designing and implementing an analysis, the key to good performance and precision is the choice of an appropriate heap model [Kanvar and Khedker 2016]. To make a client analysis precise, it typically requires so-called *strong updates*, which discard relationships that no longer hold at the current program location. Performing a strong update requires *must-alias* information about *all* aliases of a given pointer dereference [Lhoták and Chung 2011]. In Figure 1, for instance, to

infer that the call to `close` (line 4) closes not just `a` but also `b`, the analysis must know that both variables must-alias. To track must-alias information, instead of tracking pointers individually, current static analyses [Fink et al. 2008; Naeem and Lhoták 2011] track entire *sets* of pointers through the program. Unfortunately, the resulting powerset-abstraction is relatively inefficient, causing current must-alias analyses to scale badly.

To overcome these deficiencies, we present IDE^{al}, an alias-aware extension to the existing dataflow analysis framework *Interprocedural Distributive Environment (IDE)* [Sagiv et al. 1996]. IDE^{al} transparently extends IDE with a reusable, precise, and efficient heap model that includes handling of aliasing in an optimal integrated manner. To optimize for precision, IDE^{al} supports strong updates as well as a flow-sensitive and context-sensitive analysis. To optimize for performance, IDE^{al} computes alias relationships in a demand-driven, client-context-dependent manner. Further, to the best of our knowledge, IDE^{al} is the first framework that resolves must-alias relationships *without* resorting to expensive-to-compute powerset abstractions. With IDE^{al}, we show that it is indeed possible to compute must-aliasing correctly even when efficiently tracking pointers individually, not as sets of pointers.

IDE^{al} can be useful wherever objects or their fields' contents must be tracked, in particular to implement any kind of field and flow-sensitive analysis. We specifically showcase IDE^{al}'s practical relevance, its precision and soundness by instantiating an IDE^{al}-based typestate analysis (TS^{al}) and empirically evaluate it in comparison to TS^{f}, a highly precise and efficient state-of-the-art typestate analysis by Fink et al. [2008]. Our experiments show that the heap model in IDE^{al} reduces the amount of relevant dataflow facts to be propagated by a factor of 10.4×. In line with that, TS^{al} analyzes 10.3× fewer methods compared to TS^{f}. Despite IDE^{al} computing aliases on-demand, and TS^{f} relying on a pre-computed whole-program pointer analysis, TS^{al} is as efficient and precise as TS^{f} even in the worst case, where it is applied to large benchmarks as a whole.

In a case study we discuss IDE^{al} as the underlying framework of an analysis that detects incorrect usages of the Java Cryptographic Architecture (JCA) library. The JCA is a frequently used cryptographic library that ships with any standard Java installation. The analysis instantiates IDE^{al} as both a taint and a typestate analysis. In experiments on 200 Android applications, the IDE^{al}-based analysis terminates within two minutes per application on average, showing that IDE^{al} can be used to implement also feature-rich static analyses efficiently. Our results further confirm earlier studies showing that few applications use cryptography correctly [Egele et al. 2013].

In summary, this paper makes the following contributions:

- We present IDE^{al}, an extension to the IDE framework that resolves aliases automatically, efficiently, and precisely.
- We provide a practical solution to perform sound strong updates, while propagating aliases individually.
- We discuss our implementation of an IDE^{al}-based typestate analysis, and experimentally evaluate its precision and recall compared to the state of the art, and assess its performance at runtime.
- We present a full implementation of a client analysis, a crypto usage-checker, which uses IDE^{al} both for taint and typestate analysis.

2 BACKGROUND

This section presents background information about the main building blocks of IDE^{al}: the IDE framework, its better-known historical predecessor IFDS, and the alias analysis Boomerang.

2.1 The Original IFDS Algorithm

The algorithm for solving Interprocedural Finite Distributive Subset (IFDS) problems [Reps et al. 1995] is an efficient fixed-point algorithm that can be used to define a flow- and context-sensitive dataflow analysis. IFDS has three inputs, a *finite dataflow-domain D*, a *supergraph* (an *interprocedural control-flow graph*), and *flow functions* $f : S \times D \to \mathcal{P}(D)$ that transform a single dataflow fact $d \in D$ before a statement $s \in S$ to a set of facts that hold after the statement. In IFDS, flow functions are of four different types. *Normal-flow functions* specify the transformation of dataflow facts at non-call statements. At call sites, the *call-to-return-flow function* propagates dataflow facts at the side of the caller. The *call-flow function* maps dataflow facts from the caller's scope to those of the potential callees. The *return-flow function* maps dataflow facts at exit points of a callee to the successor statements of the original call site. Internally, IFDS transforms the dataflow analysis into a reachability problem over an *exploded supergraph ESG*. A node in *ESG* comprises a dataflow fact $d \in D$ and its related statement s. Throughout the paper, we use $\langle s, d \rangle$ to refer to a node with statement s and dataflow fact d in *ESG*.

Distributive flow functions are key to the efficiency of IFDS. For any two dataflow sets $A, B \subseteq D$ and flow function f, $f(A \cup B) = f(A) \cup f(B)$ must hold. This property makes it sound and precise to propagate facts $d \in D$ individually. Non-distributive frameworks instead must always propagate entire flow sets $A \subseteq D$. IFDS is particularly efficient because distributivity enables storing point-wise procedure summaries, one for each abstract input to a function, that can be reused as soon as the matching individual fact is seen again at another call site.

2.2 The Original IDE Algorithm

Sagiv et al. [1996] extended IFDS to Interprocedural Distributive Environments (IDE). In addition to the *ESG* of IFDS, IDE computes *environments*, functions $env: D \to L$, where L is a bounded-height lattice, and D is the finite dataflow-domain of the extended IFDS instance. The environments encode mappings of dataflow elements of D to values in the lattice L. IDE expects *environment transformers* for each statement to compute the environments. The transformers are functions $t: Env(D, L) \to Env(D, L)$, where $Env(D, L)$ is the set of all environments. Similar to flow functions, the environment transformers describe the effect of a statement on the lattice value for a particular dataflow fact. IDE requires those environment transformers to be *distributive*: $(t(\sqcap_i env_i))(d) = \sqcap_i (t(env_i))(d)$ for any $d \in D$ and $env_1, env_2, \ldots \in Env(D, L)$. This property allows an environment transformer to be split and represented by *edge functions*, functions of the form $f: L \to L$ that are assigned to the edges of the *ESG*. Evaluation of the edge functions than computes the same environment as the environment transformer. Using of identity function as edge functions on every edge of the *ESG*, IDE can be used to solve IFDS problems.

Similar to IFDS, IDE is a fixed-point algorithm. During the *ESG* construction, the corresponding edge functions are composed, met and propagated, once the construction of the *ESG* is done, the resulting edge functions are evaluated to yield the final lattice values associated with each node of the *ESG*, i.e. the environment. The latter process is known as *Phase 2* of IDE.

Figure 2 shows an example that uses IDE to perform linear constant propagation [Sagiv et al. 1996]. The figure depicts the *ESG* that IDE generates during its fixed-point iteration. For this example, D is the set of local variables and the graph represents the flow functions on D as straight edges. Each dataflow fact d (the local variable) is shown at the top of the column where the node is drawn. Nodes are placed between two statements, because each node represents a fact that holds after and before a statement. To uniquely identify a node, we refer to the statement that is before it. Constant propagation starts at assignments of constant integers to variables, here at line 6. The assignment `v = u` (line 7) transfers the value of `u` to `v`. The dataflow fact `v` that holds before the call

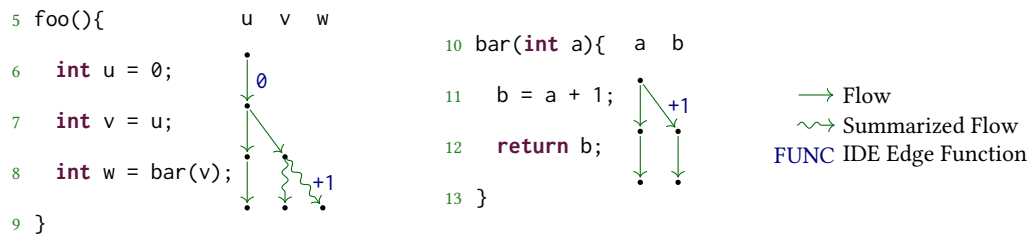


Fig. 2. Linear constant propagation modeled in IDE.

to bar (line 8) flows into callee bar. Upon termination of the analysis of bar, IDE stores the summary information that a flows to a and b. This summary is then applied at the call site to bar. In the figure, this is highlighted by the squiggled edges labeled as summarized flows. To complete the linear constant propagation, IDE adds edge functions to compute the final environment. The environment of a linear constant propagation associates to each variable a set of integer values. To the statement u = 0, IDE assigns the constant edge function $\lambda v.0$, here denoted just by 0. Within bar, the flow from a to b at the statement b = a + 1 (line 11) receives the edge function $\lambda v.v + 1$, denoted by +1. This edge function simply increases every incoming lattice value by one. IDE then promotes the edge function +1 to a summarized flow of bar. The summary of bar states that value of variable a flows to b and additionally increases the lattice value by one. Eventually, IDE composes the edge functions along each flow, making use of the summaries where appropriate. In the example, the linear constant propagation computes the environment that maps the variables u, v and a to the value 0, and b and w to the value 1.

2.3 Boomerang: An All-Aliases Analysis

In any language involving pointer variables, dataflow can occur also indirectly, through reads and writes from/to aliased memory locations. In Java this involves field reads and writes. To be sound and precise, static analyses must handle such aliasing, i.e., they should resolve which pointers may point to which memory locations. In this paper, we show that one can offload the entire tracking of aliases from the IDE^{al} framework using an all-aliases analysis. To resolve aliases internally, IDE^{al} builds on top of Boomerang [Späth et al. 2016], a demand-driven, context-sensitive and flow-sensitive pointer analysis. In contrast to traditional points-to and alias analyses [Sridharan et al. 2005; Yan et al. 2011], for a given variable v, Boomerang determines the points-to set of v, and efficiently computes *all* variables in the current scope that may point to the objects that this points-to set refers to. Boomerang thus computes *all aliases* of v. Boomerang first computes a backward pass to determine the points-to set of v, followed by a forward pass to determine all aliasing variables (in the same scope as v, optionally filtered by some calling context).

3 OVERVIEW OF IDE^{al}

We now provide an overview of IDE^{al} before explaining the internal details of the framework in Section 4. The workflow of IDE^{al} consists of two consecutive phases: the *object-flow propagation* (Phase OF) and the *value-flow propagation* (Phase VF). Both phases execute an instance of IDE, they only differ in their propagated edge functions. The propagations of both phases start at the same *seed*, a client-defined source node of the *ESG*. The seed is a node $\langle s, p \rangle$ of an abstract pointer p at a statement s.

3.1 Phase OF: Object-Flow Propagation

The purpose of Phase OF is to follow the flow of an object through the program. Instead of statically abstracting an object by its allocation site, IDE^{al} uses access graphs [Geffken et al. 2014; Khedker et al. 2007]. An access graph represents a local variable that is potentially followed by a regular expression of field accesses (e.g., `v.f+.g`). IDE^{al} uses the fully qualified field names to distinguish fields with the same names but declared in different classes. The data-flow domain D of IDE^{al} is fixed to the set of all access graphs, referred to by $\mathcal{A}$.

Given an access graph $p \in \mathcal{A}$ from the seed statement s, IDE^{al} derives all other access graphs q at any control-flow successor t of s such that q at t is an alias to p at s. In other terms, the access graph q may access the same object as p is pointing to. Tracking this flow is challenging, because a statement can introduce aliasing pointers *directly* and *indirectly*. After an assignment statement of the form `x = y`, the pointers `x` and `y` alias directly. However, at *field-write* and *call* statements, indirectly aliased access graphs may be introduced. To detect those indirectly aliased pointers, Phase OF keeps track of *points of aliasing* (*POAs*), each of which executes a pointer query. From the result of the pointer queries, IDE^{al} derives and propagates indirect pointer flows. The pointer query is required to compute *all* aliases of the pointer p at the given statement s, which IDE^{al} uses Boomerang to compute. Phase OF uses IDE and models flows of directly aliased pointers using standard pointer-flow functions (Section 4.1) that operate on $\mathcal{A}$. We modified the original IDE algorithm to keep track of *POAs* that model indirectly aliased flows automatically. As edge functions, this phase fixes all functions to be the identity edge function.[1] Due to this fact, Phase 2 of IDE is not executed in Phase OF.

By construction of our modified IDE algorithm, all nodes $\langle t, q\rangle$ that are propagated may point-to the same object as the seed node $\langle s, p\rangle$. We refer to the collection of all those nodes as the object-flow graph (*OFG*). The edges of the *OFG* originate from (1) the standard-flow functions and (2) the indirect flows at the *POAs*. The *OFG* is the *ESG* of our modified IDE problem with the fixed standard flow functions. In this work, we only use the *OFG* for visualization purposes but it is never constructed explicitly by IDE^{al}. After the fixed-point of IDE has been found, Phase OF is complete and IDE^{al} executes Phase VF.

3.2 Phase VF: Value-Flow Propagation

The goal of Phase VF is to compute IDE environments. I.e. associate a value of a client-defined finite-height lattice to each of the nodes of the *OFG*. As each node of the *OFG* is a pair of an access graph and a statement, Phase VF computes the lattice value associated to the access graph at the statement. For Phase VF, IDE^{al} triggers a second round of IDE, this time also Phase 2 of IDE is executed. Phase 2 of IDE remains unchanged in IDE^{al}. For Phase VF, IDE^{al} uses the same standard pointer flow functions and the *POAs* that are used in Phase OF. In addition to the seed, Phase VF requires the user to supply distributive environment transformers in the form of IDE edge functions that describe the transformations of the lattice values along the interprocedural path(s) of the *OFG*. Note, the environment transformers can only be represented as edge functions, if the environment transformers are distributive. This holds in general for IDE and must also be guaranteed when instantiating an analysis in IDE^{al}.

Clients that have already been shown to obey the distributivity property and therefore are instantiable in IDE^{al} are linear constant propagation [Sagiv et al. 1996], more precise dataflows in the presence of correlated calls [Rapoport et al. 2015] or the analysis of software product lines [Bodden et al. 2013].

[1] Phase OF simplifies to an IFDS problem. For the sake of the presentation, we present it as an IDE problem.

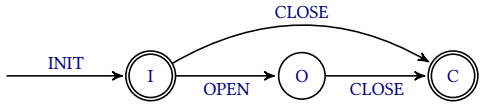


Fig. 3. The finite state machine for `File` objects.

During Phase VF, all *POAs* are known and their queries are computed. The results of the pointer queries at each *POA* determines whether or not a bypassing pointer's value is strongly updated.

Even though both IDE^{al} phases perform an IDE fixed-point computation that is only distinguished by the edge functions, IDE^{al} has to compute the IDE fixed-point twice. A necessity when one wants to perform strong updates but propagate aliased pointers separately. We discuss this in Section 3.3, where we also elaborate on a typestate analysis example instantiated in IDE^{al}.

3.3 Typestate Analysis in IDE^{al}

In this section we show how to instantiate an IDE^{al}-based typestate analysis by defining a concrete lattice L and the respective environment transformers for a typestate analysis.

3.3.1 Instantiation. Typestate properties can be encoded in finite state machines (FSM). The FSM for an object of type `File` is drawn in Figure 3. At the end of the lifetime of a `File` object, its typestate must be in an accepting state C or I. Formally, a FSM for an object of type T has the form $A := (\Sigma, S, s_0, \delta, F)$. Σ is the set of methods that may be invoked on an object of type T changing the state of the object, S the set of all possible states, s_0 is the initial state, δ is the transition function $\delta: S \times \Sigma \to S$ and F is the set of accepting states of the finite state machine. The lattice for the typestate analysis in IDE^{al} is the lattice $L_A := (\mathcal{P}(S), \cup)$, the powerset of the set of states S ordered by set union. Concretely, we map each node of the *OFG* to a set of states of the FSM.

The dataflow domain D in IDE^{al} is fixed to the set of all access graphs $\mathcal{A}$ and the environment transformers concretizes to $t: Env(\mathcal{A}, L_A) \to Env(\mathcal{A}, L_A)$. We are now able to provide the environment transformers and show their distributivity. The state of an object of type T may only change at a call site that invokes a method $m \in \Sigma$ on a receiver of type T. For any other statement, the environment transformer is identity. Assume the base variable on which the call is invoked to be a. For such a call $a.m$, the environment transformer $t_{a.m}$ has the form $\lambda env.env[a \mapsto \{\delta(s,m) \mid s \in env(a)\}]$. [2] That is, the set of states $env(a)$ associated with the access graph a *before* the statement is replaced by the set of states that is constructed by performing the transition m on each of the states within $env(a)$. Any lattice value associated with any other access graph than a is maintained.

We now show that the environment transformer for each statement is distributive. Assume $env_1, env_2, \ldots \in Env(\mathcal{A}, L_A)$ are concrete environments and $d \in \mathcal{A}$. Without loss of generality, one can restrict the environment transformer to be of the form $t_{a.m}$ and d to be the access graph representing the local variable a. In all other cases, the transformers are the identity functions that obey distributivity. Then it holds that:

$$\begin{aligned}\sqcap_i(t_{a.m}(env_i))(d) &= \sqcap_i\{\delta(s,m) \mid s \in env_i(d)\} = \cup_i\{\delta(s,m) \mid s \in env_i(d)\} \\ &= \{\delta(s,m) \mid s \in \cup_i env_i(d)\} = \{\delta(s,m) \mid s \in \sqcap_i env_i(d)\} = (t_{a.m}(\sqcap_i env_i))(d)\end{aligned}$$

This proves that the environment transformers are distributive and can be represented by edge functions $f : L_A \to L_A$ on top of the *OFG* edges. Hence, in the remaining of the paper we will refer to the edge functions instead of the environment transformers. It is important to mention that the

[2] The notation $\lambda env.env[a \mapsto S]$ means, that the lattice value associated to the access graph a is the set of states S and any other access graph maintains its values.

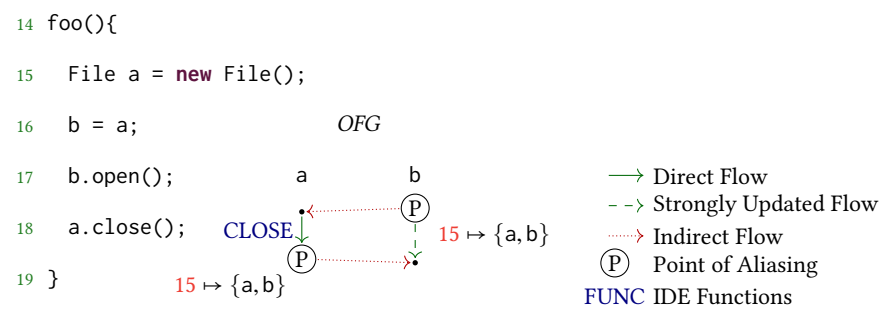


Fig. 4. An example illustrating how an IDE^{al}-based typestate analysis checks for unclosed files.

environment transformers for the typestate analysis change only the typestate of the access graph on which the call is invoked, but do *not* change the typestate of any other aliasing access graph that refers to the same object. This suffices because IDE^{al} associates the resulting lattice values with all those aliasing access graphs automatically, by solving the appropriate *POAs*.

3.3.2 Example. Figure 4 shows an example of applying an IDE^{al}-based typestate analysis that checks for unclosed files within the program, i.e. the FSM drawn in Figure 3. The typestate analysis starts off at the seed node $\langle 17, \texttt{b}\rangle$, a potential source of a typestate violation as the `File` object may remain open. Based on the seed, IDE^{al} starts the propagation in Phase OF and generates all the nodes associated with `b` in the *OFG*. During this propagation, IDE^{al} registers $\langle 17, \texttt{b}\rangle$ as the first *POA*. IDE^{al} then issues an all-aliases query to Boomerang for `b` at line 17, yielding $\{15 \mapsto \{\texttt{a}, \texttt{b}\}\}$. This result means that `b` at line 17 points to the object allocated at line 15, and that $\{\texttt{a}, \texttt{b}\}$ are the *only* pointers in scope at line 17 that point to this object. In other words, `a` is an alias to `b`. Therefore, IDE^{al} adds the indirect-flow edge from `b` to `a` to the *OFG*. This newly created node unveils a new part of the *OFG* that contains the two nodes associated with `a`.

IDE^{al} then discovers another *POA* $\langle 18, \texttt{a}\rangle$ on the return from the call[3] to `close` (line 18). The results of the pointer query is the same as before and the second indirect-flow edge from $\langle 18, \texttt{a}\rangle$ to $\langle 18, \texttt{b}\rangle$ is added to the graph. Since $\langle 18, \texttt{b}\rangle$ has already been propagated, IDE^{al} does not discover any new nodes in the *OFG*, and it concludes the computations in Phase OF. In the final *OFG*, the node $\langle 18, \texttt{a}\rangle$ encodes that, after statement 18, `a` may point to the same object that `b` points to at statement 17. IDE^{al} now executes Phase VF to propagate user-specific values from a lattice L along the final *OFG*.

IDE^{al} expects an initial lattice value that is associated with the seed node for Phase VF. In the example in Figure 4, we assign the singleton set with the state {O} of the FSM with the node $\langle 17, \texttt{b}\rangle$. The object referred to by `b` is in an open state at this point. The typestate analysis assigns the edge function f_{18} = CLOSE to the *OFG* edge from $\langle 17, \texttt{a}\rangle$ to $\langle 18, \texttt{a}\rangle$. This edge function holds the CLOSE transitions $\{\text{I} \to \text{C},\ \text{O} \to \text{C}\}$ of the FSM in Figure 3. Eventually, IDE^{al} composes and evaluates the edge functions along all possible interprocedural paths of the *OFG*. The initial lattice element {O} at the seed node flows via the indirect-flow edge from `b` to `a` into the function f_{18}. Applying f_{18} to the input {O} yields the state {C}. The lattice value {C} then flows via the indirect-flow edge back to `b`, and IDE^{al} infers that `b` *may* be closed. To determine that the object *will* in fact be closed, IDE^{al} must prevent the opened lattice element represented by {O} from bypassing the call to `close`.

[3]For simplicity, we do not show the code of `close()` here, but IDE^{al} will analyze it.

Since IDE^{al} can infer that a and b must-alias, the flow from $\langle 17, \texttt{b}\rangle$ to $\langle 18, \texttt{b}\rangle$ that bypasses the call to `a.close()` is removed in Phase VF. The killed flow is shown by the green dashed edge in the OFG of Figure 4. We say the flow is strongly updated. The strong update prevents the propagation of the stale information that b remains opened. We will discuss how IDE^{al} determines must-alias relations in Section 4.3. After the call `a.close()`, the lattice value associated with $\langle 17, \texttt{b}\rangle$ and $\langle 18, \texttt{a}\rangle$ is the singleton set {C}. Since {C} is an accepting state of the FSM, both a and b point to objects that are eventually closed.

IDE is a chaotic fixed-point iteration, and the order in which nodes of the *ESG* are created is non-deterministic. In Phase VF of IDE^{al}, flows are killed depending on the existence of other nodes in the graph. In the example, the value on variable b that bypasses the call `a.close()` is strongly updated only if the *POA* $\langle 18, \texttt{b}\rangle$ has been detected. Therefore, the iteration order is relevant for IDE^{al}, which explains our design decision to execute two consecutive IDE phases.

4 FRAMEWORK DESIGN

In this section, we explain the details of the main algorithm in IDE^{al}. We define the standard flow functions in IDE^{al}, and describe how we handle points of aliasing, perform sound strong updates, and support context-sensitive alias queries.

As explained in the previous section, IDE^{al} uses a modified version of IDE that keeps track of *POAs*. To achieve that, IDE^{al} intercepts the IDE fixed-point iteration to detect indirect flows of pointers at points of aliasing, which are special nodes of the *OFG*. Depending on the node, additional indirect flow edges are created and injected into the worklist of IDE.

We base our description on the pseudo-code provided for the standard IDE algorithm by Sagiv et al. [1996, p. 147]. The IDE algorithm maintains and extends path edges that describe the (summarized) intraprocedural realizable flows within the *OFG*. We write a path edge as $d_1 \rightarrow \langle s, d_2\rangle$, the dataflow fact d_1 is the intraprocedural dataflow element that the *OFG* node $\langle s, d_2\rangle$ originates from and is called the *source fact*.

The fixed-point iteration of IDE is controlled by a function called Propagate [Sagiv et al. 1996]. Figure 5 depicts the Propagate function and highlights the required changes for IDE^{al}. Due to space restriction, we omit the full IDE algorithm here. In addition to the main worklist called *PathWorkList*, IDE stores jump functions in the map *JumpFn*. The jump functions map each path edge to its edge function that describes how a lattice value is transformed along the path edge. IDE relies on the jump functions to decide if its fixed-point has been reached.

Within Propagate, we introduce two changes. First, a node of the *OFG* can be a point of aliasing. In such a case, the necessary aliases are computed by a pointer analysis. The call to computeAliases (line 3) abstracts this construction, because the concrete aliases depend on the type of *POA*. For each alias d_3 of d_2, a path edge with the same source fact d_1 is created and propagated along with the jump function f that reached node $\langle s, d_2\rangle$. The second change (lines 8-10) affects the propagation during Phase VF. Some of the points of aliasing introduce strong updates. If a dataflow fact is strongly updated, i.e., it receives the value of another aliased pointer, the flow is killed and the subsequent propagation is prevented.

Strong updates can only be performed in Phase VF, once *all* necessary alias queries have already been performed (in Phase OF) and their results are known. This enables IDE^{al} to propagate aliasing pointers in a distributive manner, while maintaining strong updates.

4.1 Standard Flow Functions

IDE^{al} uses a set of fixed standard flow functions to model the direct pointer flow at statements. The interprocedural-flow functions are straightforward. For each access graph whose base variable

```
1:  procedure PROPAGATE(d1 → ⟨s, d2⟩ ,f)
2:      if isPointOfAliasing(⟨s, d2⟩) then
3:          aliases = COMPUTEALIASES(d1 → ⟨s, d2⟩)
4:          for d3 ∈ aliases ∧ d3 ≠ d2 do
5:              PROPAGATE(d1 → ⟨s, d3⟩,f)              ▷ Indirect flows through aliases
6:          end for
7:      end if
8:      if Phase VF ∧ RECEIVESSTRONGUPDATE(d1 → ⟨s, d2⟩) then
9:          return                                     ▷ Strong update in Phase VF kills flow.
10:     end if
11:     f′ = f ⊓ JumpFn(d1 → ⟨s, d2⟩)
12:     if f′ ≠ JumpFn(d1 → ⟨s, d2⟩) then
13:         JumpFn(d1 → ⟨s, d2⟩) = f′
14:         PathWorkList.add(d1 → ⟨s, d2⟩)
15:     end if
16: end procedure
```

Fig. 5. The required changes in IDE for IDE^{al}.

$$[\![x \leftarrow new]\!](\langle\!\langle v, sEt \rangle\!\rangle) = \begin{cases} \varnothing & \text{if } v = x \\ \{\langle\!\langle v, sEt \rangle\!\rangle\} \end{cases} \tag{1}$$

$$[\![x \leftarrow y]\!](\langle\!\langle v, sEt \rangle\!\rangle) = \begin{cases} \{\langle\!\langle x, sEt \rangle\!\rangle, \langle\!\langle v, sEt \rangle\!\rangle\} & \text{if } v = y \\ \varnothing & \text{if } v = x \\ \{\langle\!\langle v, sEt \rangle\!\rangle\} \end{cases} \tag{2}$$

$$[\![x \leftarrow y.f]\!](\langle\!\langle v, sEt \rangle\!\rangle) = \begin{cases} \{\langle\!\langle v, sEt \rangle\!\rangle\} \cup \{\langle\!\langle x, G \rangle\!\rangle \mid G \in tail(sEt)\} & \text{if } v = y \wedge s = f \\ \varnothing & \text{if } v = x \\ \{\langle\!\langle v, sEt \rangle\!\rangle\} \end{cases} \tag{3}$$

$$[\![x.f \leftarrow y]\!](\langle\!\langle v, sEt \rangle\!\rangle) = \begin{cases} \{\langle\!\langle v, sEt \rangle\!\rangle\} \cup \langle\!\langle x, cons(f, sEt) \rangle\!\rangle & \text{if } v = y \\ \varnothing & \text{if } v = x \wedge s = f \\ \{\langle\!\langle v, sEt \rangle\!\rangle\} \end{cases} \tag{4}$$

Fig. 6. The intraprocedural normal-flow functions of IDE^{al}.

matches an actual parameter of the call, including the call receiver, the *call-flow function* simply replaces the base by its corresponding formal parameter. Similarly, at return sites, the *return-flow function* maps the base of an access graph back to the corresponding actual argument, while also replacing the returned variable with the assigned variable (if applicable).

IDE^{al} operates on a three-address code, an intermediate representation where statements contain at most one field dereference. Table 1 lists the statements that IDE^{al} handles, and Figure 6 provides the definitions of the corresponding intraprocedural *normal-flow* functions. Each function $[\![w]\!](\alpha)$ maps an access graph α at statement w to a set of access graphs that hold after statement w. We represent an access graph as $\langle\!\langle x, sEt \rangle\!\rangle$, where the base x is a local variable, and sEt is a *field graph*. A field graph is a directed graph whose nodes are fields and is uniquely identified by the edge set E and two special nodes, the first access s and the last access t.

Table 1. Three-addressed code that IDE^{al} handles.

Statement	Notation
Allocation site	`x ← new`
Assign statement	`x ← y`
Field read statement	`x ← y.f`
Field write statement	`x.f ← y`
If statement	`if(x ≠ null) goto l`
Call site	`m(p)`

Equation (1) in Figure 6 shows the flow function for an allocation site $x \leftarrow new$. IDE^{al} kills each access graph of the form $\langle\!\langle x, \cdot \rangle\!\rangle$, due to the re-assignment of the base x.

Equation (2) describes the flow of access graphs for a local-assignment statement $x \leftarrow y$. For access graphs of the form $\langle\!\langle y, sEt \rangle\!\rangle$, IDE^{al} generates the aliasing access graph $\langle\!\langle x, sEt \rangle\!\rangle$, which is the result of assigning y to x. Similar to allocation sites, IDE^{al} kills access graphs of the form $\langle\!\langle x, \cdot \rangle\!\rangle$. IDE^{al} preserves the flow for all other cases.

Equation (3) defines the flow function for a field-read statement $x \leftarrow y.f$. The first case handles access graphs of the form $\langle\!\langle y, fEt \rangle\!\rangle$, where the edge set E and field t are arbitrary. For each edge $(f, g) \in E$, for some field g, IDE^{al} uses the *tail* operation to compute the field graph $g\bar{E}t$, where $\bar{E} = E \setminus \{(f, g)\}$, generating the access graph $\langle\!\langle x, g\bar{E}t \rangle\!\rangle$. IDE^{al} kills access graphs of the form $\langle\!\langle x, \cdot \rangle\!\rangle$, as shown in the second case of the equation. For all other cases, IDE^{al} maintains the access graph $\langle\!\langle y, fEt \rangle\!\rangle$.

Equation (4) handles a field-write statement $x.f \leftarrow y$. IDE^{al} calls $cons(f, sEt)$ to update each access graph $\langle\!\langle y, sEt \rangle\!\rangle$ that reaches the field-write statement by making the field f its head. Formally, the *cons* operation computes the field graph $f\bar{E}t$, where $\bar{E} = E \cup \{(f, s)\}$. IDE^{al} then constructs and propagates the new access graph $\langle\!\langle x, cons(f, sEt) \rangle\!\rangle$, as well as $\langle\!\langle y, sEt \rangle\!\rangle$. At the field-write statement, IDE^{al} kills any access graph $\langle\!\langle x, fEt \rangle\!\rangle$ with an arbitrary edge set E and a last field access t, because y overwrites the field f of x.

The described standard pointer flow functions are used as defaults in IDE^{al}, but they can be adjusted by the client. For example, a taint analysis may model flows through string concatenations.

4.2 Points of Aliasing

IDE^{al} resolves points of aliasing by issuing alias queries to Boomerang [Späth et al. 2016] and uses the result: *all* aliasing access graphs for a given pointer dereference. We denote a points-to query that IDE^{al} issues to Boomerang by $BR(s, \alpha)$, and it comprises a statement s and an access graph α. The result of the query is a set of *pointer elements*. A pointer element has the form (t, A_t), where t is an allocation site and A_t is the set of all access graphs β for which $t \in points\text{-}to(\beta)$ holds at s. The access graph α, for which the query is issued, is an element of the set A_t. We write $\beta \in BR(s, \alpha)$ when α and β may alias at s. That means, there exists a pointer element (t, A_t) such that $\beta \in A_t$.

We now discuss which nodes of the *OFG* are points of aliasing and the aliases they compute. A *POA* can either be [Write], [Return], or [Null], and depending on this type, the computed aliases differ. In terms of the algorithm of Figure 5, we describe the result of the function computeAliases. In the following, let us assume $d_1 \rightarrow \langle s, d_2 \rangle$ to be the path edge that is currently propagated, i.e., the argument to computeAliases.

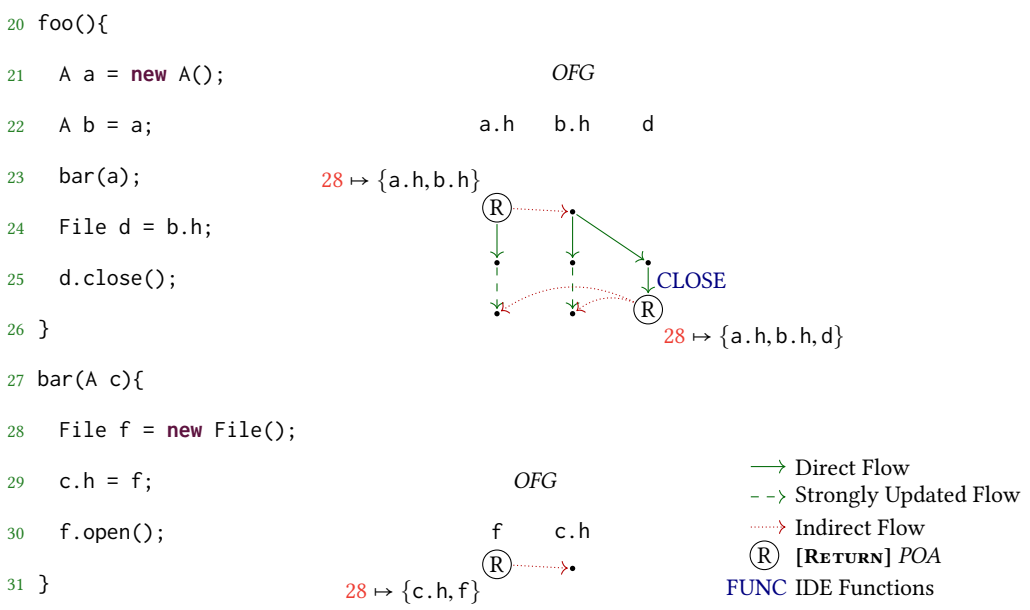


Fig. 7. An example illustrating how IDE^{al} resolves multiple **[Return]** *POAs* and performs sound strong updates.

4.2.1 ***[Write].*** A **[Write]** is a node $\langle s, d_2 \rangle$ in the *OFG*, where s is a field-write statement $x.f \leftarrow y$, and d_2 is an access graph $\langle\!\langle x, fEg \rangle\!\rangle$ with arbitrary E and last access g. Each alias of x generates an aliasing access graph. IDE^{al} computes those aliases by first issuing the query $BR(s, \langle\!\langle x, \varnothing \rangle\!\rangle)$ to compute all aliases of the access graph $\langle\!\langle x, \varnothing \rangle\!\rangle$ before s. IDE^{al} then concatenates the field graph fEg to each aliasing access graph $\langle\!\langle z, vFw \rangle\!\rangle \in BR(s, \langle\!\langle x, \varnothing \rangle\!\rangle)$. This creates the new access graphs $d_3 = \langle\!\langle z, conc(vFw, fEg) \rangle\!\rangle$. The operation *conc* yields the field graph $v\bar{E}g$, where $\bar{E} = E \cup F \cup \{(w, f)\}$. The set of aliases that is returned by *computeAliases* for a **[Write]** is the set of all d_3s. For each d_3, IDE^{al} then injects the new path edge $d_1 \rightarrow \langle s, d_3 \rangle$. When we draw the *OFG*, we highlight the injection of such an edge as an indirect-flow edge from $\langle s, d_2 \rangle$ to $\langle s, d_3 \rangle$.

4.2.2 ***[Return].*** A **[Return]** is a node $\langle s, d_2 \rangle$ in the *OFG*, where s is a method call $m(p)$, and d_2 is an access graph $\langle\!\langle p, fEg \rangle\!\rangle$. When the callee returns the access graph to a call site, there might be local aliasing access graphs in the caller's scope that point-to the same abstract object. For example, an alias to p may exist prior to the call site s, and instead of using $\langle\!\langle p, fEg \rangle\!\rangle$ to access the object of interest, the aliased pointer may be used.

Similar to the treatment of **[Write]**, IDE^{al} computes all aliases to the pointer $d_2 = \langle\!\langle p, fEg \rangle\!\rangle$, denoted by $d_3 \in BR(s, d_2)$. IDE^{al} then adds indirect-flow edges from $\langle s, d_2 \rangle$ to each node $\langle s, d_3 \rangle$ by propagating $d_1 \rightarrow \langle s, d_3 \rangle$. As the algorithm in Figure 5 shows, the same edge function that is propagated to $\langle s, d_2 \rangle$ is also used for this indirect aliasing path edge with target node $\langle s, d_3 \rangle$. By doing so, the additional indirect-flow edges in Phase VF maintain the lattice value flow from d_2 and propagate the value to each alias d_3 after the call. In the case of a typestate analysis, this value flow transfers a possible state change on the access graph $\langle\!\langle p, fEg \rangle\!\rangle$ within the callee to all of its aliasing access graphs within the caller. For example, in Figure 7, at the call to open (line 30),

IDE^{al} starts tracking the pointer `f` that references a `File` object and the pointer then flows into the callee `open`. At the return to `bar`, IDE^{al} registers a **[Return]** that yields $[28 \mapsto \{\texttt{c.h}, \texttt{f}\}]$ and adds an indirect-flow edge from $\langle 30, \texttt{c.h}\rangle$ to $\langle 30, \texttt{f}\rangle$. Since `c` is a parameter of `bar`, the access graph `c.h` escapes to `foo` via the call site at line 23. Since `c.h` corresponds to `a.h` in `foo`, IDE^{al} creates the node $\langle 23, \texttt{a.h}\rangle$ and registers it as another **[Return]** that yields $[28 \mapsto \{\texttt{a.h}, \texttt{b.h}\}]$. IDE^{al} then adds the appropriate indirect-flow edge to the *OFG*. Without this indirect-flow edge, IDE^{al} misses that the tracked object is loaded from `b.h` into to `d` in the next statement. After the call to `close` on `d` (line 25), IDE^{al} registers another **[Return]** that yields $[28 \mapsto \{\texttt{a.h}, \texttt{b.h}, \texttt{d}\}]$ and adds indirect-flow edges to the nodes $\langle 25, \texttt{a.h}\rangle$ and $\langle 25, \texttt{b.h}\rangle$.

4.2.3 [Null]. A **[Null]** is a node in the *OFG* that corresponds to a comparison to `null` in the program. To avoid throwing a `NullPointerException` in Java, it is common to check for *nullness* of variables before accessing them. These checks lead to branches in the control flow. IDE^{al} handles **[Null]** to avoid imprecise propagations along the branches where IDE^{al} knows that the object can never be `null`. We discuss this in an example in the next subsection.

4.3 Sound Strong Updates

During Phase VF, IDE^{al} performs sound strong updates for the client's analysis information. To achieve that, dataflows are killed in Phase VF, as shown by the early termination of the Propagate that depends on the call to receivesStrongUpdate (Figure 5, lines 8-10). A created node of the *OFG* receives a strong update iff there is an indirect flow to that node from a *POA* and the *POA*'s pointer query returned a unique pointer allocation. In such situations, the access graphs in question can only point to a single *abstract* object, and thus must point to the same one. For example, in Figure 7, IDE^{al} labels the initial seed node $\langle 30, \texttt{f}\rangle$ with the initial lattice element {O}, because the tracked object is initially in an OPEN state. This lattice element flows along all the *OFG* paths (as they are labeled by the identity edge function) and reaches the call to `close` at line 25. At this call, it is sound to perform a strong update for the aliases `a.h` and `b.h`, because **[Return]** at $\langle 25, \texttt{d}\rangle$ yields a single allocation site. Since `a.h` and `b.h` are not involved in the call `d.close()`, but abstract the same object, IDE^{al} kills the call-to-return flows in Phase VF, i.e. those path edges receive a strong update. This correctly prevents the O state from bypassing the call. Instead, IDE^{al} transfers the typestate information from $\langle 25, \texttt{d}\rangle$, via the indirect-flow edges, to the nodes $\langle 25, \texttt{a.h}\rangle$ and $\langle 25, \texttt{b.h}\rangle$, and an IDE^{al}-based typestate analysis will report that the `File` object pointed to by `a.h` and `b.h` is eventually closed.

The condition to perform a strong update ("singleton points-to set") is insufficient in cases where the allocation site is within a loop or a recursive part of the program. Such re-entrant allocation sites may represent multiple concrete runtime objects by a single abstract object. To ensure soundness for these cases, IDE^{al} configures the underlying pointer analysis to treat assignments of the form `x = null` as allocation sites as well. Due to the semantics of Java, a `null`-assignment outside the loop must exist, and the pointer analysis will pick it up, causing IDE^{al} to avoid an otherwise incorrect strong update. Figure 8 shows a code snippet with an allocation site inside a loop (line 35). At line 36, the file is opened by a call to `a.open()`. In the `if` branch, the file is assigned to `b`. When the branches join again, `b.close()` is called. When `b` returns from the call to `close`, IDE^{al} registers a **[Return]**, computing all aliases to `b` at line 40: $[33 \mapsto \{\texttt{b}\}, 35 \mapsto \{\texttt{a}, \texttt{b}\}]$. The pointer `b` may point to two different abstract objects, making a strong update of the pointers to these objects unsound. IDE^{al} therefore does *not* strongly update the call-to-return flow to node $\langle 40, \texttt{a}\rangle$. Following the flow path and tracking the object states, the typestate analysis computes that the object that `b` points to is definitely closed. However, the object pointed to by `a` is either closed or opened. This result

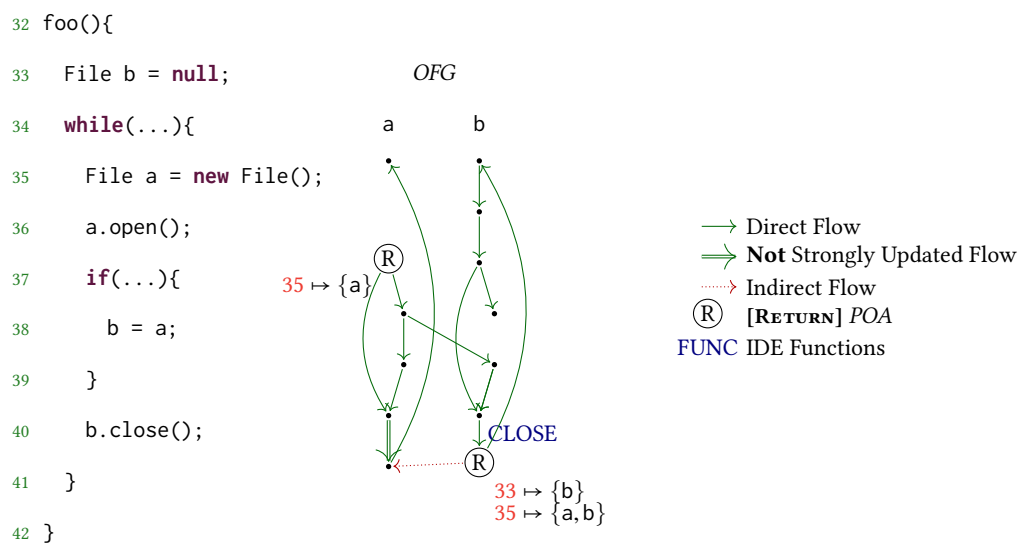


Fig. 8. An example illustrating that IDE^{al} does not perform unsound strong updates in loops.

matches the possible runtime execution where b may refer to an object from an earlier iteration of the loop and a currently points to a different object that is not closed.

Strong updates are also used at [Null] to prevent flows along invalid branches. If IDE^{al} creates an *OFG* node $\langle s, \langle\!\langle x, \varnothing \rangle\!\rangle\rangle$ with a statement $s : \mathit{if}(x \neq \mathit{null})$, it registers a [Null] and issues an alias query $BR(s, \langle\!\langle x, \varnothing \rangle\!\rangle)$. If the access graph $\langle\!\langle x, \varnothing \rangle\!\rangle$ has only one allocation site that differs from `null`, all computed aliasing access graphs that bypass the condition check are also different from `null`. Therefore, IDE^{al} avoids the propagations of these pointers to the branch where x is `null`. In Phase VF, IDE^{al} strongly updates the path edges with targets $\langle t, \beta \rangle$, where t is the `false`-successor of the if statement and $\beta \in BR(s, \langle\!\langle x, \varnothing \rangle\!\rangle)$. We showcase this behavior in the example in Section 5.

4.4 Context-Sensitive Alias Queries

While IDE^{al} inherits its context-sensitivity from IDE, Boomerang is a context-sensitive pointer analysis by design. To achieve optimal precision and performance, IDE^{al} matches, for each alias query, the contexts it provides to Boomerang to its own IDE contexts. IDE achieves context-sensitivity by maintaining the source fact for each node created in the *OFG*, which is why IDE propagates path edges instead of the plain *OFG* nodes. IDE^{al} consults the source fact and the *incoming* relationship of IDE/IFDS [Naeem et al. 2010] to match the call site(s) once an *OFG* node reaches a return site. IDE^{al} supplies Boomerang with the currently considered IDE contexts to limit its alias queries to these given contexts. For example, when IDE^{al} issues an alias query for an access graph that originates from some parameter to the current method, it limits the scope of Boomerang to search for allocation sites only within the callers of the current method that the IDE solver visited, but *no other callers*. Considering other callers may yield spurious aliases that are impossible along any actual runtime execution path of the program. In the following section we give an example of the explained context-matching.

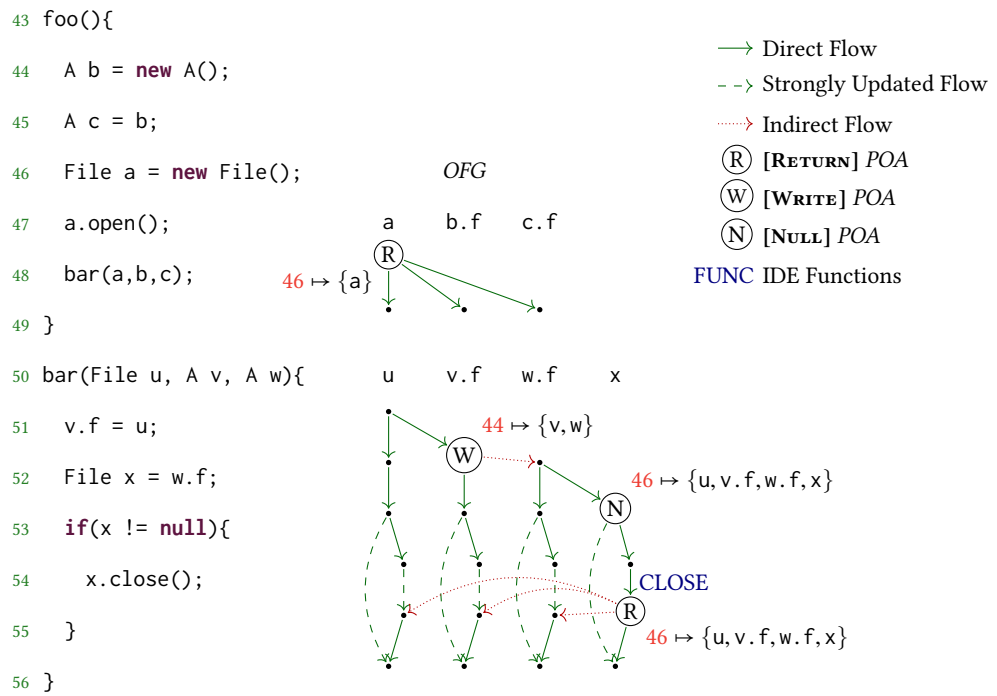


Fig. 9. A complete run of IDE^{al}.

5 COMPLETE RUN

Figure 9 shows an example of a complete run of IDE^{al}. In the example, the `File` object that is opened at line 47 represents the seed that the client provides. This seed flows as an argument to `bar` (line 48) and is assigned to `v.f` (line 51), registering a [Write] at the node $\langle 51, \texttt{v.f}\rangle$. The source fact of this *POA* is `u`, because it is the intraprocedural origin of the propagation. IDE^{al} then uses Boomerang to compute the aliases of `v` at line 51, but limited to the calling contexts that initiated the propagation. The source fact `u` of the *POA* is a parameter of `bar` and during the pointer query for variable `v` a calling context of `bar` is requested by Boomerang. IDE^{al} uses the incoming relationship of IDE to limit the scope of Boomerang to only the call site 48, the call site where the abstract pointer `u` entered `bar`. Therefore, the result of the alias query is $[44 \mapsto \{\texttt{v}, \texttt{w}\}]$. This filtering of calling contexts avoids computing aliases if there were any other callers of `bar`.

Due to the new aliasing pointer `w`, an indirect flow to `w.f` occurs at the field write, and IDE^{al} adds an indirect-flow edge from $\langle 51, \texttt{v.f}\rangle$ to $\langle 51, \texttt{w.f}\rangle$ to the *OFG*. The derived transitive flows discover two more *POAs*: a [Null] at node $\langle 52, \texttt{x}\rangle$ and a [Return] at node $\langle 54, \texttt{x}\rangle$. These *POAs* trigger two all-alias queries for variable `x`. For both queries, the source fact is again `u`, and when Boomerang requests more calling context, it is still the call to `bar` at line 48. Eventually, both queries yield the result $[46 \mapsto \{\texttt{u}, \texttt{v.f}, \texttt{w.f}, \texttt{x}\}]$. The [Return] creates the dotted indirect-flow edges that originate from $\langle 54, \texttt{x}\rangle$ in Figure 9. Since in both cases, for [Return] and for [Null], the size of the points-to

set is one, IDE^{al} performs a strong update. All other flows with the same target *OFG* nodes are killed in Phase VF.

Finally, in Phase VF, IDE^{al} computes the lattice values for all *OFG* nodes. Following all existing paths in the constructed *OFG*, IDE^{al} deduces that the last action performed on each pointer is the call to `close`. Therefore, the abstract object allocated at line 46, which is accessible via `a`, `b.f`, and `c.f` in `foo` and `u`, `v.f`, `w.f`, and `x` in `bar`, is always closed at the end of its lifetime. This matches the runtime behaviour. If IDE^{al} did not handle [Null] *POAs*, it would have used the identity function for the `else` branch within `bar`, and incorrectly deduced that the tracked `File` object may remain open. However, IDE^{al} treats the `null` check precisely, inferring that the file object cannot possibly bypass the call to `close`.

6 EVALUATION

We evaluate IDE^{al} through a typestate client analysis. We refer to this client by TS^{al} and compare it to the typestate analysis by Fink et al. [2008], referred to by TS^{f}. TS^{f} is a staged analysis where less precise stages restrict the overhead for more sophisticated analyses in later stages. All stages are based on the IFDS framework. The last stage of TS^{f} has a similar precision to TS^{al}, which enables us to directly compare the IFDS propagation statistics of TS^{f} to that of TS^{al}. Another important difference between the two analysis is their heap model. To group aliased pointers and track their shared typestate, TS^{f} uses a powerset abstraction as their dataflow domain. Our experiments empirically evaluate the characteristics of IDE^{al} by addressing the following research questions:

- **RQ1**: What is the effect of the heap models TS^{al} and TS^{f} on the generated *ESG*?
- **RQ2**: How does TS^{al} perform on large programs when compared to TS^{f} in terms of computation time?
- **RQ3**: How precise are the analysis results reported by TS^{f} and TS^{al}?
- **RQ4**: What impact do aliasing and strong updates have on TS^{al}?

In addition to those research questions, this section completes with a case study that shows how IDE^{al} can be used to realize an efficient and precise analysis that detects insecure cryptographic API usages in Android applications.

6.1 Setup

Our framework IDE^{al}, and hence TS^{al}, is based on Soot[4] and relies on the IDE solver Heros[5]. The analysis TS^{f} is publicly available[6] and is based on WALA[7].

Both typestate analyses can verify typestate properties that are encoded as state machines. Table 2 lists the typestate properties that we want to enforce. The corresponding state machines are available as part of TS^{f} analysis. TS^{al} maps them to the respective edge functions for IDE^{al}.

For all experiments, the analyses pre-compute a 0-1-CFA call graph for the given programs. For each program, we also analyze its library dependencies, including the Java Runtime Library. Given that TS^{f} is built on top of WALA and TS^{al} is built on top of Soot, we carefully configured Soot and WALA to enable a fair comparison for both analyses. All experiments are run on a modern laptop computer with a 2.9 GHz Intel Core i7 processor running Java 1.8.0_73. For each run, we allocated a maximum of 6 GB of JVM heap space.

[4] https://github.com/Sable/soot
[5] https://github.com/Sable/heros
[6] https://github.com/tech-srl/safe
[7] http://wala.sourceforge.net/

Table 2. Typestate properties represented in the micro benchmark by Fink et al. [2008].

Name	Description
`Vector`	Never try to retrieve an element of an empty vector.
`Iterator`	Always call `hasNext` on an iterator before receiving the `next` element.
`URL`	Never set options on an already connected `URLConnection`.
`IO`	Do not read or write to a closed `Stream` or `Writer`.
`KeyStore`	Always initialize a `KeyStore` before using it.
`Signature`	Always follow the phases of initialization of a `Signature`.

Table 3. Comparing the efficiency of TS_2^f, TS_3^f and TS^{al} in terms of IFDS propagations and the number of visited methods.

		***ESG* Nodes**			**Visited Methods**		
Typestate	**# Programs**	TS_2^f	TS_3^f	TS^{al}	TS_2^f	TS_3^f	TS^{al}
`Vector`	30	3,451	2,058	580	66	49	13
`Iterator`	17	507	456	171	25	20	8
`URL`	2	8,510	8,488	115	308	287	4
`IO`	14	3,161	1,005	283	96	39	8
`KeyStore`	3	2,732	92	45	91	8	3
`Signature`	6	35,175	94,537	356	1,130	1,112	8

The default configuration of TS^f performs three analysis stages. During our initial experiments, we discovered that the first stage does not report any typestate violation, preventing any computation of subsequent stages. We consulted with the authors of TS^f who confirmed this behaviour and were unable to fix the problem. Therefore, to compute meaningful results for TS^f, we deactivated the first stage. The remaining two stages are the *Unique Verifier* (TS_2^f) and the *APFocus* (TS_3^f) [Fink et al. 2008] that we base our evaluation on.

6.2 RQ1: Heap Models

In the heap model of TS_3^f, each dataflow element consists of (1) the *allocation site* of the tracked instance, (2) a *set of must-aliased pointers* to that allocation site, (3) a *completeness* flag indicating whether the set of must-aliased pointers is complete, and (4) the *state* the object is currently in.

To inspect the differences between the heap models of TS^f and TS^{al}, we ran both analyses on a set of micro benchmarks that consists of 72 sample programs. These programs ship with the implementation of TS^f. The micro benchmarks contain typestate violations with aliasing and strong update scenarios that challenge typestate analysis. Hence, it is a good baseline for the comparison.

We have applied both TS^f and TS^{al} to check for the typestate properties in Table 2 in the micro-benchmark programs. Table 3 summarizes our findings. In the table, the columns for *ESG nodes* gather the number of *OFG* nodes in TS^{al}, respectively the number of nodes in the exploded supergraph for TS^f. *Visited Methods* is the number of different methods those nodes belong to, i.e. visited by the underlying solvers. All numbers are arithmetic means taken over all input programs (column *# Programs*) of the micro benchmark. Table 3 shows the statistics for TS^f according to the

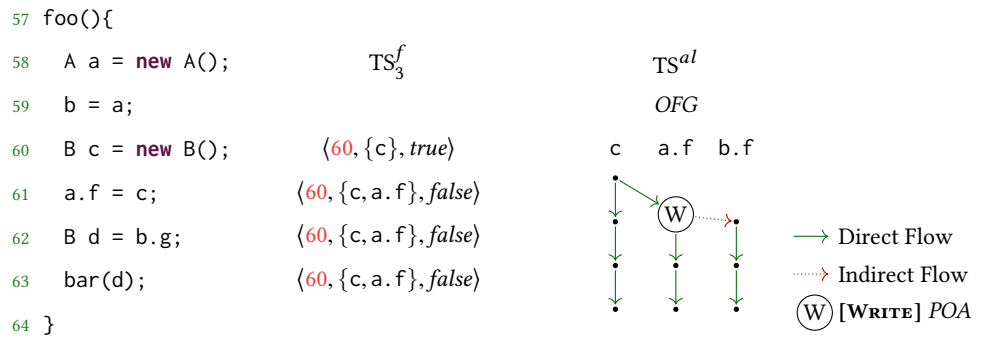


Fig. 10. An example illustrating the differences between TS_3^f and TS^{al} with respect to the structure of dataflow facts.

two analysis stages TS_2^f and TS_3^f. If TS_2^f proves that the program is error free, TS_3^f is not invoked. On the benchmark, in 14 out of the 72 programs TS_3^f was not invoked.

Since TS_3^f is the most precise stage in TS^f, we only compare TS^{al} to this stage in the following discussion. The numbers for TS_2^f are similar. Across all the micro-benchmark programs, TS_3^f requires a geometric mean of 10.4× more nodes, and analyzes approximately 10.3× more methods compared to TS^{al}.

For the typestate property `KeyStore`, stage TS_3^f only creates 92 nodes. For two of the three `KeyStore` programs, the stage TS_2^f proves the absence of a typestate violation and the stage TS_3^f is never executed. For the typestate properties `URL` and `Signature`, TS^{al} requires less than 1.4% and 0.4% of the propagations that TS_3^f requires, respectively. For `URL`, TS^{al} starts from the call sites to the method `connect` of any `URLConnection` object and reports an error once a method that sets an option on the object is invoked. In contrast, TS_3^f starts earlier at the allocation site of the `URLConnection` itself. TS_3^f records aliases only during forward propagation, while TS^{al} gets the automatic support from IDE^{al} to detect aliases before the seeds by issuing the appropriate alias queries to Boomerang. IDE^{al} evaluates those queries on demand, requiring fewer propagations. The same reasoning applies to the typestate property `Signature`.

Table 3 shows that there is a big difference in the number of created nodes in TS^f and TS^{al}. This difference originates from the heap models that the underlying framework of each analysis uses. Each element of the dataflow domain in IDE^{al} consists of an access graph. The access graph's base is a local variable associated with a declaring method. An access graph has only to be propagated within that declaring method. The dataflow abstraction used by TS^f cannot make use of this additional information. Figure 10 illustrates an example. We ignore the propagated typestate property to simplify the example. Assume both TS^{al} and TS_3^f track the object that is created at line 60. After line 61, TS_3^f propagates the abstraction $\langle 60, \{\texttt{c}, \texttt{a.f}\}, \textit{false}\rangle$. The *completeness* flag is set to *false* because after the field write statement, the tracked object is also accessible via the pointer `b.f`, which is not in the set of musted-alias pointers ($\{\texttt{c}, \texttt{a.f}\}$). Since the representation does not explicitly store `b.f`, TS_3^f has to assume that the tracked object could also be accessed in method `bar` (called at line 63), although no appropriate pointer ever escapes to the method. Therefore, TS_3^f propagates the dataflow fact $\langle 60, \{\texttt{c}, \texttt{a.f}\}, \textit{false}\rangle$ into `bar`, needlessly increasing the number of *ESG*

Table 4. Analysis time for running TS^{al} and TS$^{f}_{3}$ on the DaCapo benchmark programs.

	Vector							IO						Iterator	
	ANTLR	BLOAT	CHART	ECLIPSE	LUINDEX	LUSEARCH	PMD	ANTLR	BLOAT	CHART	ECLIPSE	LUINDEX	LUSEARCH	BLOAT	PMD
Seeds	2	6	1	10	12	1	10	1	12	4	1	1	1	14	1
Average Time per Seed (s) TS^{al} / BOOMERANG / TS$^{f}_{3}$	4.9 / 5.3	11.3 / 18.4	2 / 7.4	4.2 / 2.7	7 / 5.8	30 / 5.9	30 / 16.7	1.3 / 3.7	23.3 / 27.7	30 / 29.3	2.9 / 3.5	2.5 / 4	2.7 / 5.2	15.2 / 20.5	1.2 / 5.9
Timeouts	- -	2 2	- -	1 -	1 -	1 -	10 2	- -	9 8	4 3	- -	- -	- -	7 6	- -

nodes. On the other hand, TS^{al} does not propagate any dataflow facts into `bar`, as the object of interest cannot be accessed from variable `d`, the only variable that escapes to `bar`. The *OFG* drawn in Figure 10 shows that no node for variable `d` is ever created. TS^{al} completely skips the analysis of `bar`.

> When compared to TS^{f}, the more precise dataflow facts of IDE^{al} and their individual propagation of aliased pointers enables TS^{al} to analyze smaller, yet relevant, parts of the program.

6.3 RQ2: Performance

In this research question, we compare the analysis time performances of TS^{f} and TS^{al} on larger, real world programs. We setup both analyses with programs from the 2006 DaCapo benchmark suite [Blackburn et al. 2006] and run all of the typestate properties from **RQ1** on the programs. We exclude the benchmark FOP from our evaluation as it lacks some dependencies that are required to perform static analysis. On the remaining benchmarks, we executed both analyses and compare the results on basis of the exact same seeds. Given the different analysis frameworks TS^{f} and TS^{al} depend on, the call graphs they use are not completely identical, and not exactly the same parts of the program are reachable. As seeds, both analysis only consider allocation statements of objects of interest that are reachable in the underlying call graph. We report results only on the set of seeds that are consistent in both analyses. For the verification of each seed, we limit the execution time of the analysis to 30 seconds. This is necessary as we aim for analyzing the complete program, and in some cases, dataflow occurs through Java collections, such as `HashSet` or `List`, or massive `Visitor` patterns that remain hard to analyze.

For a fair comparison with TS^{f}, we configured TS^{al} to always start at allocation sites of objects. This neglects the full potential of IDE^{al}. For the typestate property `IO` for instance, it is sufficient to begin the analysis only at calls to `close` and report an error once a subsequent `read` is invoked on a `FileReader`. Based on the result from **RQ1** and the fact that any of the preceding stages of TS^{f} can also be used for TS^{al}, in **RQ2** we limit the comparison to stage TS$^{f}_{3}$.

Table 4 reports the result of this experiment. We cannot report any results for the properties `URL`, `KeyStore`, and `Signature`, because the DaCapo benchmark has no uses of `KeyStore`, occurrences of `Signature`, or seeds to `URL`. The table shows the total number of seeds per benchmark program

Table 5. Comparing the reported errors of TS^{f} and TS^{al}.
[a)] Correct dataflow but no typestate violation.

	True Positives		False Positives			False Negatives		
Benchmark	TS^{f}	TS^{al}	TS^{f}	TS^{al}	TS^{-SU}	TS^{f}	TS^{al}	TS^{-al}
Micro Benchmark	42	48	1	1	+3	6	0	+2
DaCapo	11[a)]	11[a)]	0	0	0	0	0	+6

per typestate property. The table also contains a bar plot, above its bars, we depict the arithmetic means over the analysis times per seed. The reported times are averaged across five independently repeated runs. Additionally, below the bar chart, we report the number of seeds that timed out.

Except for 5 out of the 15 analyzed combinations, TS^{al} is faster than TS^{f}. TS^{al} tends to be slower when verifying the `Vector` property. On geometric average across the `Vector` typestate, TS^{al} is 1.1× slower than TS^{f}. On the other two typestate properties `IO` and `Iterator`, TS^{al} is faster by a ratio of 1.5× and 2.6×, respectively. The analysis times do not reflect the results from **RQ1** as the two analyses differ in a major point. TS^{f} relies on a points-to analysis that has been computed for the call graph. On the other hand, TS^{al}, or rather IDE^{al}, uses the on-demand alias analysis Boomerang to find aliases. The time required to process the alias queries is included in the analysis time of TS^{al}. Across all programs, the time spent to compute the alias queries represent a major part of the total analysis time (55%). In the bar chart in Table 4, we highlight the overhead time from using Boomerang in TS^{al}. Nevertheless, including the overhead of the demand-driven alias analysis time, across the DaCapo benchmarks and all typestate properties, TS^{al} is 1.3× faster than TS^{f}.

> On the DaCapo benchmark programs, including all their library dependencies, a typestate analysis based on our framework IDE^{al} is slightly more efficient than a state-of-the-art typestate analysis.

6.4 RQ3: Precision

The authors of TS^{f} have annotated all the programs of the micro-benchmark from **RQ1** with the ground truth, stating which results a precise and sound typestate analysis should report. Based on that ground truth, in Table 5, we show the three results: *true positives*, the valid expected typestate errors; *false positives*, the spurious reports that do not reflect typestate errors; and *false negatives*, missed typestate errors that should have been detected.

On the micro-benchmark from **RQ1**, TS^{f} reports one false positive on the `Vector` typestate. For this program, the flow-insensitive alias analysis in TS^{f} computes an alias set that has a spurious element, leading to imprecise results. In addition to that false positive, TS^{f} misses some flows, because it does not detect some of the typestate violations for the typestate properties `Vector`, `Iterator`, and `IO`. Further investigation shows that TS^{f} incorrectly handles array accesses and flows from thrown exceptions to the corresponding catch clause. However, we believe that this is more of an implementation issue in TS^{f} rather than a conceptual problem.

TS^{al} reports one false positive on the micro-benchmark, but does not miss any report from the ground truth results. The false positive stems from a `hasNext` call within a synchronized block. The control-flow graph that Soot creates contains edges from each statement within the synchronized block to a catch block outside the block. Due to those exceptional edges, the `hasNext` call might be missed and an error is reported.

On the DaCapo benchmark suite, we manually inspected the reported errors from the analyses performed in **RQ2** to assess precision and recall. We restricted our inspection to the seeds for which TS^{al} reported errors, but also to those that finished within the given time budget of 30 seconds. In total, 11 typestate violations are reported. All the errors are reported for the typestate property `Vector`: one is reported in ECLIPSE, the other 10 are located in LUINDEX. Further investigation shows that all of the identified and tracked objects *may* be in an error state, i.e. an element of the `Vector` may be accessed before any element was added to the `Vector`. However, our manual inspection unveiled that the accesses to the elements are guarded by appropriate size checks on the `Vector`. This means, at runtime, a typestate violation cannot occur. This shortcoming is due to the weakness of the expressiveness of the typestate pattern: the size checks cannot be modeled within the typestate machine, and is not an artifact of the over-approximations of IDE^{al} nor its dataflow domains. In other words, there are potentially valid dataflow connections that lead to those reports, but they are overcome by additional information that the analysis is not designed to track. Therefore, we classify the reported errors as true positives (with respect to the tracked typestate pattern).

An IDE^{al}-based typestate analysis is as precise as an existing typestate analysis.

6.5 RQ4: Effect of Aliasing and Strong Updates

In addition to comparing both analyses, we now discuss the impact of handling aliasing and strong updates for the typestate client analysis TS^{al}. Based on the same setup used in the earlier research questions, we run TS^{al} with two additional configurations. For one run, we ignore the alias queries completely (denoted by TS^{-al}). In the other configuration, typestate updates are not strongly updated (denoted by TS^{-SU}). Both configurations allow us to compare the effect of aliasing and strong updates on TS^{al}. In general, turning off strong updates introduces additional false positives, but does not impact the number of false negatives. Strong updates only kill spurious typestate flows. In contrast, in the configuration TS^{-al}, when alias queries are disabled, false negatives are expected as the dataflow path(s) may not be complete.

Table 5 additionally depicts the analysis when run in the configurations TS^{-al} and TS^{-SU}. Column TS^{-SU} lists the false positives that are added when strong updated are disabled, and TS^{-al} lists the false negatives that are missed when aliasing is disabled. In addition to the original false positive that TS^{al} reports, disabling strong updates causes TS^{-SU} to report three additional false positives for the micro-benchmark. On the micro-benchmark, aliasing is only required for the `Vector` programs where dataflow occurs through fields. Therefore, two of the ground truth results require the typestate analysis to soundly handle aliasing.

The results of this experiment on DaCapo differ slightly. When we disable aliasing for TS^{al}, more than 50% of the errors are not reported anymore. For all of those cases, the tracked objects are stored inside fields of other objects and are accessed indirectly within other methods through the fields. Such dataflows require aliasing information about the parent object, the object that the field resides within, information that is missing when aliasing is disabled. On the other hand, disabling strong updates on DaCapo programs does not report any false positives on the inspected seeds.

While strong updates marginally improve precision on the micro-benchmarks, on the DaCapo programs, handling aliases has a higher influence and is required to obtain sound results.

6.6 Case Study: A CryptoAnalyzer Built on Top of IDE^{al}

In the preceding research questions, we evaluated IDE^{al} by a qualitative comparison of an IDE^{al}-based typestate analysis to the analysis TS^{f} implemented in WALA. We further want to demonstrate

```
65 public class Encrypter{
66   private SecretKey key;
67   private int keyLength = 448;
68
69   public Encrypter(){
70     KeyGenerator keygen = KeyGenerator.getInstance("Blowfish");
71     keygen.init(this.keyLength);
72     this.key = keygen.generateKey();
73   }
74
75   public byte[] encrypt(String plainText){
76     Cipher cipher = Cipher.getInstance("AES");
77     cipher.init(Cipher.ENCRYPT_MODE, this.key);
78     byte[] encText = cipher.doFinal(plainText.getBytes());
79     return encText;
80   }
81 }
```

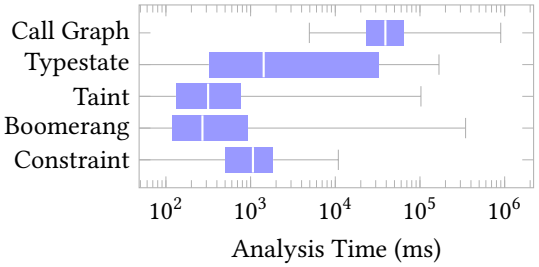
Fig. 11. An example of a broken cryptographic implementation.

the strength of IDE^{al} by applying it to another concrete client analysis, an analysis that verifies the correct usage of the Java Cryptographic Architecture (JCA). Cryptography is a complex topic on its own. In combination with an ambiguous API design of the JCA, multiple researchers have shown that cryptography is commonly implemented incorrectly [Egele et al. 2013; Nadi et al. 2016]. Improperly used cryptography frequently leads to data leaks.

Figure 11 demonstrates an example of a misuse of a cryptographic cipher that originates from the `javax.crypto.Cipher` class of the JCA. The code snippet shows a class that is supposed to be used for encryption. When an object of type `Encrypter` is instantiated, the constructor generates a `SecretKey` of size 448 for the Blowfish algorithm. Then in line 72, the generated key is stored as field `key` of the constructed instance. Calling the `encrypt` method creates a `Cipher` object in line 76, but for the AES algorithm. The next line initializes the algorithm's mode to encryption and supplies the `SecretKey` that is stored in the field. The `doFinal` operation in the next line performs the encryption of the `plainText`.

There are three issues in this code example. First, the generated key and the encryption cipher do not match (AES vs. Blowfish). Second, and related, the key length of 448 is not suitable for an AES algorithm that expects a size of 128. Third, depending on the crypto provider, AES is used with electronic codebook mode (ECB) which results in low entropy of the bytes of the encoded ciphertext `encText`. While the first two misuses throw runtime exceptions, and thus are likely detected during development, the latter silently leads to insecure code.

Using IDE^{al}, we implemented a static-analysis tool that detects those and similar issues for the Java cryptographic API. On a high level, the analysis is composed of the typestate analysis TS^{al}, additional queries to an extended version of Boomerang which extract `String` and `int` parameters on-the-fly, an IDE^{al}-based taint analysis (i.e., edge functions are identity) and a constraint-solving analysis. In the example code, the crypto analysis starts at the two calls to `getInstance` and ensures with TS^{al} that the `init` and `generateKey`, respective `init` and `doFinal` methods are invoked in that order on the seed objects referenced by the variables `keygen` and `cipher`. The integer value 448 for `this.keyLength` is extracted through a query to Boomerang.



Specification	# Misuses
`MessageDigest`	139
`Cipher`	8
`PBEKeySpec`	3
`PBEParameterSpec`	1

Fig. 12. The distribution of the analysis times separated into the different subanalyses of our crypto analysis and the classes ranked by the number of misuses.

The call sequence on the `KeyGenerator` object `keygen` satisfies the required typestate automaton. The analysis further infers that an abstract object, a SecretKey for the algorithm Blowfish with a key length of 448 has been generated at line 72. The object flows (taint analysis) to the field `this.key` and eventually into the `init` method of the `Cipher`. The `Cipher` object also satisfies the typestate, but combining the algorithm parameter of the `Cipher` object, namely AES and the corresponding parameter of the `SecretKey`, Blowfish, leads to a contradiction when the crypto analysis tries to solve the encoded constraints.

In addition to the aforementioned classes, we specified rules for a total of 17 classes of the JCA that drive the crypto analysis. Those rules include classes like `KeyStore`, `PBEKeySpec` for password-based encryption and `SecretKey`, their interactions and suggested choice of parameters.

We run the analysis on a set of 200 Android applications that we downloaded from Andro-Zoo [Allix et al. 2016]. All applications within the set were distributed via the Google PlayStore and received an update in 2016 or 2017. Out of the 200 applications, a total of 144 use classes of the JCA. We report our results within this subset. Figure 12 summarizes the analysis results. We divide the analysis time into the different subanalyses. The box plot shows, in logarithmic scale: the median, the lower and upper quartile (the box) that limit 50% of the data around the median, and the lower and upper whiskers (i.e., the minimal and maximal analysis times). The analysis times vary vastly across the different applications because they vary greatly in size.

Using call graphs constructed by FlowDroid [Arzt et al. 2014], the number of methods that are call-graph reachable range from 524 to 16,282, with an average of 3,715. Across the 144 applications, our crypto analysis requires an average (arithmetic mean) of 123.8 seconds to terminate. The box plot shows that most of the analysis time is spent in call graph construction (on average 63.1% of the total analysis time), followed by the typestate-analysis time. The number of seeds for the typestate analysis varies from 2 to 46 across all 144 applications.

Despite the fact that the chosen 144 applications have been recently updated, our analysis finds only 5 applications that correctly use the JCA API. The table in Figure 12 shows that most of the misuses are related to `MessageDigest`. The class is misused in 139 applications, mainly due to using either MD5 or SHA-1, both of which are no longer recommended for the purpose of cryptography.

We manually verified the correctness of some of the findings reported by the analysis. In particular, we discovered security-related typestate errors in three applications. After the usage of a `PBEKeySpec`, the developers should call `clearPassword()` to overwrite the internal copy of the password. Our analysis correctly identifies the missing calls. The issue that is listed for `PBEParameterSpec` in the table in Figure 12 concerns the number of times a hashing algorithm is applied to a password in a password-based encryption. The chosen iteration count is set to 100 but is supposed to be higher (our specification expects at least 1000). Additionally, we found some software-quality related

typestate errors. For example, in one application, a developer consecutively calls the method `init` three times with the same parameters on a retrieved `Cipher` object. In a couple of other applications, we discovered that developers (unnecessarily) reset `MessageDigest` objects after their initialization.

Our findings confirm previous studies that have reported that 88% of Android applications do not correctly make use of the JCA API [Egele et al. 2013]. Our case study shows that IDE^{al} can be used to implement practical, feature-rich client analyses efficiently in a broader context.

6.7 Threats to Validity

The evaluation we performed is restricted to a typestate analysis as it is the most related work that we could find an implementation for. There is a variety of other clients, such as linear constant propagation or analysis of correlated calls, that IDE^{al} can be used for but we have not evaluated them yet. It is interesting to see the performance of IDE^{al} on other clients, such as a linear constant propagation, but we consider it as future work. Therefore, the results give only an idea of IDE^{al}'s general performance. Finally, TS^{f} and TS^{al} are based on two different analysis frameworks, WALA and Soot, that do not operate on the exact same program representations, and have different implementations for the same standard points-to and call graph analyses.

7 RELATED WORK

In this section, we discuss existing dataflow frameworks, most of which expose the problem of aliasing to the client analysis, as well as *solutions to aliasing* that client analyses may apply. For a more extensive discussion of the state-of-the-art alias analyses for object-oriented programs, we refer the reader to the survey by Sridharan et al. [2013].

7.1 Dataflow Analysis Frameworks

Apart from IFDS [Reps et al. 1995] and IDE [Sagiv et al. 1996], there exists a wide range of dataflow frameworks that simplify the implementation of interprocedural static dataflow analyses. For example, TVLA [Sagiv et al. 1999] uses abstract predicates that evaluate to a three-valued logic. In addition to 0 (*false*) and 1 (*true*), three-valued logic maintains a third value (1/2) that represents *unknown* or *maybe* evaluations. Using predicates enables TVLA to infer aliasing automatically. TVLA has been extended later to support interprocedural analysis [Gotsman et al. 2006; Jeannet et al. 2004]. While TVLA propagates sets of aliasing pointers, IDE^{al} propagates aliasing pointers independently, which drastically reduces the size of the analysis domain.

Separation logic defines another technique for dataflow analysis [Reynolds 2002]. In separation logic the heap is modeled explicitly. Each instruction directly operates on the model of the heap, analog to an actual execution. Separation logic extends standard Hoare logic by adding a separating conjunction. The conjunction splits the heap into disjunct regions. At a callsite, for instance, the heap can be divided into the region accessed within the callee and the region's complement. With an extension called bi-abduction, Calcagno et al. [2009] managed to design a scalable shape analysis based on separation logic. IDE^{al} propagates abstract pointers (to heap cells) instead of a model of the cells on the heap, and it is not required to split the heap. Calcagno et al. [2009] achieve scalability by performing a compositional analysis. Similar benefits are expected for IDE^{al} by pre-computing summaries, as described by Arzt and Bodden [2016].

Blackshear et al. [2015] propose a goal-directed approach, called Hopper, to analyze programs that are based on event-driven systems, such as Android. The novelty lies in jumping along control-flow feasible paths, once a dataflow flows into system code, e.g. Android. Instead of analyzing the system's code, the flow directly jumps to the respective point in the non-system part of the program. Hopper relies on separation logic, i.e. explicitly models the heap. The authors report a

significant performance boost through jumping. In future work, we want to investigate how IDE^{al} can make use of a similar functionality.

Ferrara [2014] proposes an abstract-interpretation-based generic framework to value-flow analysis that includes heap-reasoning. The authors formally prove that heap and value analyses can be combined into one analysis, similar to IDE^{al}. Two types of analyses that can be instantiated within their framework are numerical and shape analysis. Opposed to their work, IDE^{al} computes context-sensitive results. This, however, comes by the cost of restricting the value domain from a infinite to a finite height lattice, as IDE^{al} does not support a widening operator.

Madsen and Møller [2014] describe a sparse dataflow analysis framework for JavaScript code. A *sparse* dataflow analysis operates on def-use chains that it constructs from the control-flow graph of a given program. This approach is different from a traditional dataflow analysis that processes every statement in the program. Sparse dataflow analyses leverage the fact that def-use chains are typically more sparse than the control-flow graph, thus the analysis requires fewer propagations of dataflow facts. IDE^{al} follows a similar approach by operating on the object-flow graph that is a sparse abstraction of the control-flow graph.

7.2 Solutions to Aliasing

We have already compared to the typestate analysis by Fink et al. [2008] in detail and skip its discussion here.

Yahav and Ramalingam [2004] propose a typestate analysis on top of TVLA, but the authors report later that TVLA does not scale well to large programs [Fink et al. 2006]. An interesting contribution of their work, however, is *separation*, as they report a huge benefit in separating the typestate analysis into subproblems. We use a simple version of separation in TS^{al} by invoking IDE^{al} per tracked object.

Naeem and Lhoták [2008] show how to perform a typestate analysis (TS^{n}), using property specifications called tracematches [Bodden et al. 2008]. Tracematches are a language extension to AspectJ [8], and allow the analysis to select program points using declarative patterns called *pointcuts*. TS^{n} uses a coarse-grained field-insensitive abstraction for the objects that are allocated on the heap. In contrast to TS^{f} and TS^{al}, TS^{n} can check for patterns that detect buggy interactions of multiple objects (e.g., updating a list while an iterator iterates over it). TS^{n} implements this by tracking all objects that are allocated in the input program, which is a significant limitation to efficiency. Naeem and Lhoták [2011] later overcome this limitation by generating flow-insensitive callee and caller summaries. The summaries are constructed by pre-analyzing the methods where pointcuts have no matches. TS^{n} then plugs in those summaries at the appropriate call sites. We plan to extend IDE^{al} to use a similar approach to synchronize information about multiple interacting objects.

Tripp et al. [2013] propose Andromeda, an IFDS-based taint analysis that handles aliasing by propagating access paths individually. Similar to IDE^{al}, Andromeda resolves aliases in a context-sensitive and flow-sensitive fashion through an on-demand backward analysis. Unlike IDE^{al}, due to propagating aliases individually, Andromeda does not support strong updates. Once tainted, Andromeda does not *untaint* an object if it is sanitized through an alias. FlowDroid [Arzt et al. 2014] takes a similar approach to alias resolution.

8 CONCLUSION

We have presented IDE^{al}, an extension to the IDE framework that automatically handles aliases in a precise and efficient manner. One of the key features in IDE^{al} is performing sound strong updates, which require must-alias information, while propagating aliases individually. This individual

[8] https://eclipse.org/aspectj/

propagation of pointers enables IDE^{al} to reuse fine-grained procedure summaries, improving its efficiency. IDE^{al} relieves static-analysis authors of the burden of encoding alias information in the dataflow domain. Using IDE^{al} enables static-analysis authors to easily implement a wide range of dataflow analyses that track object flows, such as typestate analysis and property inference by simply defining the IDE edge functions, while aliases are handled internally by IDE^{al}.

Our empirical evaluation shows that an IDE^{al}-based typestate analysis is as efficient and precise as a state-of-the-art typestate analysis, despite the overhead time of the underlying on-demand alias analysis. IDE^{al} achieves that by the use of a more precise abstraction of dataflow facts that avoids unnecessary over-approximations. This results in analyzing only relevant parts of the program. In our experiments, an IDE^{al}-based typestate analysis requires 10.4× fewer dataflow propagations than a state-of-the-art typestate analysis. On larger programs, the fewer propagations lead to a 1.3× faster typestate analysis.

ACKNOWLEDGMENTS

This research was supported by a Fraunhofer Attract grant as well as the Heinz Nixdorf Foundation. This material is also based upon work supported by the National Sciences and Engineering Research Council of Canada.

REFERENCES

Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: collecting millions of Android apps for the research community. In *International Conference on Mining Software Repositories (MSR)*. 468–471.

Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. 2005. Synthesis of interface specifications for Java classes. In *Symposium on Principles of Programming Languages (POPL)*. 98–109.

Steven Arzt and Eric Bodden. 2016. StubDroid: automatic inference of precise data-flow summaries for the android framework. In *International Conference on Software Engineering (ICSE)*. 725–735.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Programming Language Design and Implementation (PLDI)*. 259–269.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 169–190.

Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2015. Selective control-flow abstraction via jumping. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 163–182.

Eric Bodden, Reehan Shaikh, and Laurie J. Hendren. 2008. Relational aspects as tracematches. In *International Conference on Aspect-Oriented Software Development (AOSD)*. 84–95.

Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPL$^{\text{LIFT}}$: statically analyzing software product lines in minutes instead of years. In *Programming Language Design and Implementation (PLDI)*. 355–364.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Symposium on Principles of Programming Languages (POPL)*. 289–300.

Nurit Dor, Michael Rodeh, and Shmuel Sagiv. 2000. Checking Cleanness in Linked Lists. In *International Symposium on Static Analysis (SAS)*. 115–134.

Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 73–84.

Pietro Ferrara. 2014. Generic Combination of Heap and Value Analyses in Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 302–321.

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective typestate verification in the presence of aliasing. In *International Symposium on Software Testing and Analysis (ISSTA)*. 133–144.

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 2 (2008).

Manuel Geffken, Hannes Saffrich, and Peter Thiemann. 2014. Precise Interprocedural Side-Effect Analysis. In *International Colloquium on Theoretical Aspects of Computing (ICTAC)*. 188–205.

Rakesh Ghiya and Laurie J. Hendren. 1996. Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C. In *Symposium on Principles of Programming Languages (POPL)*. 1–15.

Alexey Gotsman, Josh Berdine, and Byron Cook. 2006. Interprocedural Shape Analysis with Separated Heap Abstractions. In *International Symposium on Static Analysis (SAS)*. 240–260.

Bertrand Jeannet, Alexey Loginov, Thomas W. Reps, and Shmuel Sagiv. 2004. A Relational Approach to Interprocedural Shape Analysis. In *International Symposium on Static Analysis (SAS)*. 246–264.

Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 29:1–29:47.

Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. 2007. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 1 (2007).

Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. In *Symposium on Principles of Programming Languages (POPL)*. 3–16.

Magnus Madsen and Anders Møller. 2014. Sparse Dataflow Analysis with Pointers and Reachability. In *International Symposium on Static Analysis (SAS)*. 201–218.

Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through hoops: why do Java developers struggle with cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 935–946.

Nomair A. Naeem and Ondrej Lhoták. 2008. Typestate-like analysis of multiple interacting objects. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 347–366.

Nomair A. Naeem and Ondrej Lhoták. 2011. Faster Alias Set Analysis Using Summaries. In *Compiler Construction (CC)*. 82–103.

Nomair A. Naeem, Ondrej Lhoták, and Jonathan Rodriguez. 2010. Practical Extensions to the IFDS Algorithm. In *Compiler Construction (CC)*. 124–144.

Rohan Padhye and Uday P. Khedker. 2013. Interprocedural data flow analysis in Soot using value contexts. In *International Workshop on State Of the Art in Java Program analysis, (SOAP)*. 31–36.

Marianna Rapoport, Ondrej Lhoták, and Frank Tip. 2015. Precise Data Flow Analysis in the Presence of Correlated Method Calls. In *International Symposium on Static Analysis (SAS)*. 54–71.

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Symposium on Principles of Programming Languages (POPL)*. 49–61.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Symposium on Logic in Computer Science (LICS)*. 55–74.

Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theoretical Computer Science* 167, 1&2 (1996), 131–170.

Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 1999. Parametric Shape Analysis via 3-Valued Logic. In *Symposium on Principles of Programming Languages (POPL)*. 105–118.

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *European Conference on Object-Oriented Programming (ECOOP)*. 22:1–22:26.

Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Alias Analysis for Object-Oriented Programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. 196–232.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 59–76.

Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*. 210–225.

Octavian Udrea and Cristian Lumezanu. 2006. Rule-Based Static Analysis of Network Protocol Implementations. In *USENIX Security Symposium*. 193–208.

John Whaley, Michael C. Martin, and Monica S. Lam. 2002. Automatic extraction of object-oriented component interfaces. In *International Symposium on Software Testing and Analysis (ISSTA)*. 218–228.

Eran Yahav and G. Ramalingam. 2004. Verifying safety properties using separation and heterogeneous abstractions. In *Programming Language Design and Implementation (PLDI)*. 25–34.

Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*. 155–165.