

Extracting Object-Oriented Dataflow Communication

Radu Vanciu Marwan Abi-Antoun
 Department of Computer Science
 Wayne State University
 {radu, mabiantoun}@wayne.edu

ABSTRACT

During architectural risk analysis, security experts look for architectural flaws based on a documented runtime structure, which for object-oriented systems, can be approximated by an object graph. However, extracting meaningful object graphs is challenging. For a worst-case security analysis, the object graph must be sound and show all possible objects and relations between them. It must also be hierarchical, to convey both high-level understanding and detail. Achieving soundness requires static analysis, but architectural hierarchy is not available in general programming languages.

To achieve hierarchy, we leverage ownership types and abstractly interpret the annotated program to extract a global, sound, hierarchical object graph that has dataflow communication edges showing the flow of objects due to field reads, field writes, and method invocations. We formalize the static analysis using a constraint-based specification, prove its soundness, then evaluate it, showing that the extracted edges are similar to those drawn by a security expert.

1. INTRODUCTION

For over a decade, architectural risk analysis, also known as threat modeling [22], has been one of the three pillars of building secure systems [14, Chapter 5], together with code review and penetration testing. Architectural risk analysis identifies structural problems in the design of the system, by focusing on architectural flaws rather than implementation-level defects or coding bugs. Examples of coding bugs are buffer overflow or unsafe system calls. Examples of architectural flaws are compartmentalization problems in the design or broken or illogical access control over tiers.

To look for architectural flaws, architects cannot simply read the code. They need to understand the system at a higher level. One requirement of architectural risk analysis is a documented architecture. Unfortunately, many systems that need to be analyzed do not have one. Even when the architecture is documented, it may be out of date with the

code. Reasoning about security requires an architecture that focuses on the runtime structure, rather than the code structure that shows packages and classes, or a deployment view that shows the location of the server or the firewall.

A runtime architecture shows runtime components and connectors, uses hierarchical decomposition, and partitions a system into tiers [6]. Today, the tools for runtime architectures are immature [11, 7] compared to tools for the code architecture [18]. Architectural risk analysis typically uses a Dataflow Diagram (DFD) that describes how data traverses the system, by showing data sources, external interactors, trust boundaries and processes that data goes through [24].

Moreover, a security analysis must consider the worst—rather than the typical—case of possible component communication. So the analysis results are valid only if they are based on an architecture that reveals all objects and relations that may exist at runtime—in any program run. This requires a sound static analysis to extract the architecture. In contrast, dynamic architectural extraction considers one or more program runs [19], and may miss important objects or relations that arise only in other executions.

In previous work [2], we showed how to statically extract a hierarchical runtime architecture from object-oriented code. The approach uses a sound, static analysis to extract a global, hierarchical, Ownership Object Graph (OOG), that shows all the objects across the entire system, organized hierarchically. Hierarchy conveys architectural abstraction, where architecturally significant objects appear near the top of the hierarchy and data structures are further down. An OOG is then abstracted into a standard run-time architecture, for re-documentation or conformance analysis.

In other previous work [3], we showed how an OOG relates to a DFD. There was more work to be done, however. The OOG previously showed points-to edges. During architectural risk analysis, security experts often focus on the dataflow communication between components. Extracting this communication requires additional program analysis.

Contributions. In this paper, we propose a static analysis to extract a hierarchical object graph with usage edges that show dataflow communication. Our contributions are:

- Formalizing the analysis using a constraint-based specification and proving its soundness;
- Evaluating our analysis on an extended example by comparing an OOG with dataflow edges to a DFD drawn by a security expert, and to an OOG with points-to edges.

Outline. The rest of this paper is organized as follows. Section 2 defines dataflow communication and reviews own-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

ership domains. Section 3 describes the challenges of statically extracting an object graph. Section 4 formalizes our analysis. Section 5 evaluates the analysis on an extended example. Section 6 discusses related work.

2. BACKGROUND

We define dataflow communication following Spiegel [20]. We then review ownership domains and introduce a small running example.

Definition of dataflow communication: *Let a and b be two objects. Dataflow communication exists from a to b if a reads or writes to b 's fields or calls b 's methods.*

In object-oriented code, a dataflow communication between two references a and b corresponds to field writes, field reads, or method invocations. Since the flow can be bidirectional, we distinguish between *import* and *export* dataflow.

Import dataflow communication: *An import dataflow communication exists from the source b of type B to the destination a of type A if a receives data from b . That is, there is a method ma of A such that ma refers to $b.f$ or uses the result returned by a method mb of B .*

Export dataflow communication: *An export dataflow communication exists from the source a of type A to the destination b of type B if one of b 's field f may be modified when one of a 's methods is invoked. That is, there is a method ma of A such that ma contains the statement $b.f = c$ or $b.mb(c)$, where c is in the scope of ma , i.e., a field of A , an argument of ma , an object instantiated by ma , or an object returned by another method invoked by ma .*

2.1 Review of Ownership Domains

An *ownership domain* is a named conceptual group of objects that may alias. An object is in exactly one domain that does not change at runtime, and one object can have several domains to own its substructure. Domains are declared on a class but are treated like fields, in the sense that fresh domains are created for each instance of that class. For a domain D declared on a class C and two instances $o1$ and $o2$ of type C , the domains $o1.D$ and $o2.D$ are distinct, for distinct $o1$ and $o2$.

Domains define two kinds of object hierarchy. A public domain provides *logical containment* and makes an object conceptually *part of* another. Having access to an object gives the ability to access objects inside all its public domains. A private domain provides *strict encapsulation* and makes an object strictly *owned by* another. Then, a **public** method cannot return an alias to an object inside a private domain, although the Java type system allows returning an alias to a field marked as **private**.

Domain parameters propagate ownership information of objects outside the current encapsulation, and allow objects to share state with each other. When instantiating a class that takes parameters, the client code supplies actual domains for the formal domain parameters.

Finally, a few special annotations add expressiveness to the type system. An object that is marked as **SHARED** may be aliased globally but may not alias non-**SHARED** references, and little reasoning can be done about it.

Language extensions vs. annotations. For readability in this paper, we use the language extensions of Ownership Domains [4]. Our concrete system uses available language support for annotations, which leads to verbose code con-

structs, but enables adding annotations to legacy code, after the fact, in order to reverse-engineer OOGs. The annotations still implement the type system, and a typechecker checks that the annotations are consistent with each other and with the code. For a detailed discussion of the annotation language, see [1, Appendix A].

2.2 Running Example

To illustrate the above definitions, we use a small system that follows a Document-View architecture, and which we refer to as Listeners.

Code structure. The code consists of several classes and uses various base classes, as is common in object-oriented code. The entry point of the application is the **Main** class, which instantiates **Model**, **PieChart**, and **BarChart**. **Model** extends the **Listener**, and contains the information displayed in the charts. **BarChart** and **PieChart** extend the **BaseChart** abstract class, which subsequently extends **Listener**. **BaseChart** and **Model** have a field of type **List** that represents a collection of objects of type **Listener** (Fig. 2). An instance of **Model** exchange messages of type **MsgMtoV** and **MsgVtoM** with instances of **BarChart** and **PieChart**.

Notation An object labeled $obj:T$ indicates a reference obj of type T , which we then refer to either as the “object obj ” or the “ T object” to mean “an instance of the T class.” To distinguish between domains with the same name D , we refer to a domain as $obj.D$, where obj is the parent object.

Ownership domain annotations To express the Document-View architecture, **Main** defines two domains **DOC** and **VIEW**. The object **model** is in **DOC**, while the objects **barChart** and **pieChart** are in **VIEW**. Next, **Listener** has a declaration of a public domain **DATA** for messages. **Model** and **BaseChart** declare the private domain **OWNED** for collections of **Listener** objects registered for notification. **BarChart** and **PieChart** inherit the **OWNED** domain from **BaseChart**. As a public domain, **DATA** gives access to messages, while the collections in **OWNED** are strictly encapsulated.

Objects and domains. **DATA** is inherited from the abstract class **Listener** by all its subclasses. Therefore, the OOG shows three distinct domains: **barChart.DATA**, **pieChart.DATA**, and **model.DATA**. Similarly, the OOG shows three distinct domains: **barChart.OWNED**, **pieChart.OWNED**, and **model.OWNED**. Since each object is in exactly one domain, the OOG shows three distinct **List** objects, where two of them correspond to one **new** expression in **BaseChart**.

Dataflow Communication Edges. The dataflow edges in the OOG correspond to field reads, field writes, and method invocations. An export dataflow edge exists from **main** to **model** due to the method invocation **model.addListener(barChart)**, and the edge is labeled with the type of the argument, in this case **BarChart**. An export dataflow edge exists from **model** to **barChart** and to **pieChart** due to the method invocation **l.update(mTOv)**. In the code the type of the receiver l is **Listener**; however, the OOG shows only the objects in **VIEW** as destinations. The OOG also shows two export dataflow edges corresponding to the field write **listeners.value = l**. The source is **model**, the destination is **listeners1**, while the labels are **BarChart** and **PieChart**, respectively. The import dataflow edges from **listeners1** to **model** correspond to the field read **listeners.value** in the **Model** class. Similarly to the export dataflow edges the edge labels are **BarChart** and **PieChart**.

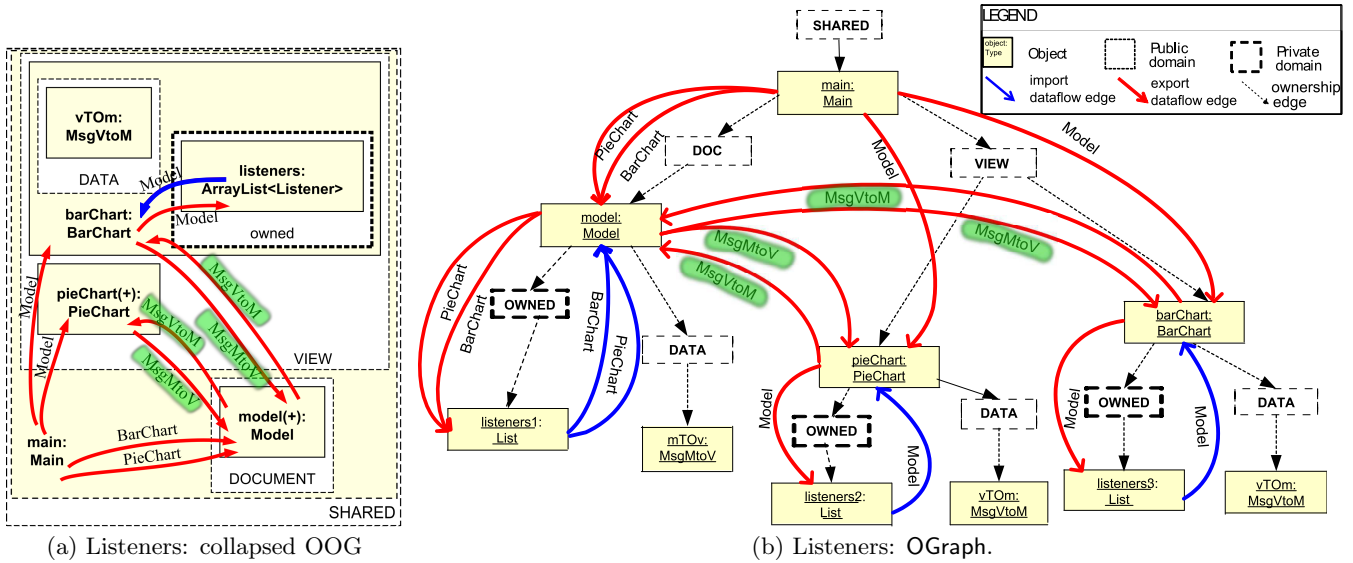


Figure 1: Listeners OOG. Interesting edges are highlighted.

Discussion. The object graph conveys architectural abstraction by organizing objects hierarchically: the architecturally significant objects such as `barChart`, `pieChart` and `model` are at a higher level of the hierarchy, and low-level objects such as collections and messages are at a lower level (Fig. 1(b)). The resulting object graph makes visually obvious the dataflow communication occurring in the program. For example, the object graph shows two dataflow edges labeled `MsgMtoV` from `model` to `barChart` and `pieChart`, and two edges labeled `MsgVtoM`, from `barChart` and `pieChart` to `model` (the highlighted edges in Fig. 1(a)). If `VIEW` and `MODEL` correspond to tiers, the above edges are interesting to highlight to an architect because they show data transferred across tiers. An OOG with points-to edges [2] will not show this communication since `model` does not have a field of type `BarChart` or `PieChart`.

Collapsed OOG. Having a hierarchical representation allows expanding or collapsing the substructure of selected objects to control the level of visual detail. For example, only the substructure of `barChart` is visible, while the substructures of `pieChart` and `model` are collapsed (Fig. 1(a)). A (+) symbol indicates that an object has a collapsed substructure. While collapsing, the visualization also lifts the parent-child dataflow edges to the nearest visible ancestor, which makes the graph less cluttered. That is, if there is an edge between low-level objects that are elided, the edge is recursively lifted to the nearest visible ancestor. The nested box visualization allows us to collapse an object’s substructure and reduce the number of objects at the top level [1, Section 3.4]. We also show the underlying representation because we explain how to extract it.

3. CHALLENGES

We first discuss the challenges of extracting object graphs statically. At runtime, the structure of an object-oriented program can be represented as a Runtime Object Graph (ROG), where nodes represent objects, i.e., instances of classes, and edges represent relations between objects, such as one object calling another object’s methods. A sound static analysis extracts an object graph that approximates

```

class Main<OWNER> {
    public domain DOC, VIEW;
    BarChart<VIEW, DOC> barChart = new BarChart();
    PieChart<VIEW, DOC> pieChart = new PieChart();
    Model<DOC, VIEW> pieChart = new PieChart();
    void run(){
        model.addListener(barChart); //(mainBarChart→model)
        model.notifyObservers(); //no dataflow
    }
}

class BarChart<OWNER, M> extends BaseChart<OWNER, M> {}
class PieChart<OWNER, M> extends BaseChart<OWNER, M> {}
class BaseChart<OWNER, M> extends Listener<OWNER> {
    domain OWNED;
    List<OWNED, Listener<M>> listeners = new List();
}

abstract class Listener<OWNER> {
    public domain DATA;
    abstract void update(Msg<DATA> msg);
}

class Model<OWNER, V> extends Listener<OWNER> {
    domain OWNED;
    List<OWNED, Listener<V>> listeners = new List();
    void addListener(Listener<V> l) {
        //(modelBarChart→listeners1, modelPieChart→listeners1)
        listeners.value = l;
    }
    void notifyObservers() {
        MsgMtoV<DATA> mTOv = new MsgMtoV();
        //(listeners1BarChart→model, listeners1PieChart→model)
        Listener<V> l = listeners.value;
        //(modelMsgMtoV→barChart, modelMsgMtoV→pieChart)
        l.update(mTOv);
    }
}

class List<OWNER, T<ELTS>> { //generic type T
    T<ELTS> value; //ELTS is a domain parameter
}

```

Figure 2: Listeners code fragments. The full code is in the companion technical report [25].

all possible ROGs, for any program execution. We represent the extracted object graph as an OGraph, where nodes are

OObjects and edges are OEdges. An OObject is a canonical object that represents multiple runtime objects. Similarly, an OEdge is a canonical edge that represents runtime dataflow communication between the corresponding runtime objects. An OGraph has the following requirements:

Object soundness. The OGraph must show a unique representative for each runtime object. While one OObject can represent multiple runtime objects, the same runtime object cannot map to two separate OObjects. It would be misleading to have one runtime entity appear as two boxes (two components) on an architectural diagram. Then one could assign the two components different values for a key `trustLevel` property and potentially invalidate the analysis results.

Aliasing. The static analysis must soundly handle possible aliasing in the program by enforcing the unique representatives invariant. For two variables in the program that may alias and refer to the same runtime object, the analysis must create a single OObject.

Edge soundness. If there is a runtime dataflow communication between two runtime objects, the OGraph must show an OEdge between the representatives of these objects.

Summarization. An ROG can have an unbounded number of runtime objects. For example, in the presence of recursive types, the ROG might have an unbounded depth. The OGraph must be a finite representation of all ROGs and must have a finite depth. The static analysis must stop creating new nodes in the OGraph at some level, and instead use already created nodes. A common heuristic is for the analysis to stop when it gets to a node of the same type as a node it previously created.

Hierarchy. A global OGraph must convey architectural abstraction by object hierarchy and support both high-level and detailed understanding of the runtime structure. It must show architecturally significant OObjects near the top of the hierarchy and OObjects representing data structures further down.

Precision. The analysis must not merge objects excessively. For example, an OGraph that represents all the runtime objects with one node is sound but very imprecise. Ideally, the OGraph must have no more OEdges than soundness requires. Like any sound static analysis, however, the OGraph may have false positives and may show OObjects or OEdges that do not correspond to a runtime object or runtime relation, due to infeasible paths in the program.

4. FORMALIZATION OF THE ANALYSIS

We present the abstract syntax, the data type declarations of the OGraph, and formally describe the most interesting static semantics of the analysis. We also describe key differences with our earlier work.

Abstract Syntax. We formally describe our static analysis using Featherweight Domain Java (FDJ), which models a core of the Java language with ownership domain annotations [4]. To keep the language simple and easier to reason about, FDJ uses Featherweight Java, which ignores Java language constructs such as interfaces and static code.

We adopt the FDJ abstract syntax (Fig. 3) but with the following changes. We exclude cast expressions and domain links, which are part of FDJ, but not crucial to our discus-

$$\begin{aligned}
CT &::= \overline{cdef} \\
cdef &::= \text{class } C \langle \overline{\alpha}, \overline{\beta} \rangle \text{ extends } C' \langle \overline{\alpha} \rangle \\
&\quad \{ \overline{dom}; \overline{T} \overline{f}; C(\overline{T'} \overline{f'}, \overline{T} \overline{f}) \\
&\quad \{ \text{super}(f'); \text{this.f} = \overline{f}; \} \overline{md} \} \\
dom &::= [\text{public}] \text{ domain } d; \\
md &::= T_R m(\overline{T} \overline{x}) T_{this} \{ \text{return } e_R; \} \\
e &::= x \mid \text{new } C \langle \overline{p} \rangle (\overline{e}) \mid e.f \mid e.f = e' \\
&\quad \mid e.m(\overline{e}) \mid \ell \mid \ell \triangleright e \\
n &::= x \mid v \\
p &::= \alpha \mid n.d \mid \text{SHARED} \\
T &::= C \langle \overline{p} \rangle \\
v, \ell, \theta &\in \text{locations} \\
S &::= \ell \rightarrow C \langle \overline{\ell'.d} \rangle (\overline{v}) \\
\Sigma &::= \ell \rightarrow T \\
\Gamma &::= x \rightarrow T
\end{aligned}$$

Figure 3: Simplified FDJ abstract syntax [4].

sion. We also include a field write expression $e.f = e'$, which can lead to dataflow communication.

In FDJ, C ranges over class names; T ranges over types; f ranges over field names; v ranges over values; d ranges over domain names; e ranges over expressions; x ranges over variable names; n ranges over values and variable names; S ranges over stores; ℓ and θ ranges over locations in a store; θ represents the value of `this`; a store S maps locations ℓ to their contents; the set of variables includes the distinguished variable `this` of type T_{this} used to refer to the receiver of a method; the result of the computation is a location ℓ , which is sometimes referred to as a value v ; $S[\ell]$ denotes the store entry of ℓ ; $S[\ell, i]$ denotes the value of i^{th} field of $S[\ell]$; $S[\ell \mapsto C \langle \overline{\ell'.d} \rangle (\overline{v})]$ denotes adding an entry for location ℓ to S ; α and β range over formal domain parameters; m ranges over method names; p ranges over formal domain parameters, actual domains, or the special domain `SHARED`; the expression form $\ell \triangleright e$ represents a method body e executing with a receiver ℓ ; an overbar denotes a sequence; the fixed class table CT maps classes to their definitions; a program is a tuple (CT, e) of a class table and an expression; Γ is the typing context; and Σ is the store typing.

Data Type Declarations. Our analysis produces a hierarchical object graph (OGraph), which has nodes representing objects and domains, and edges representing dataflow communication (Fig. 4). The OGraph is a triplet $G = \langle DO, DD, DE \rangle$, where DO is a set of OObjects, and DD maps a pair $(O, C::d)$ to an ODomain D , i.e., DD maintains a mapping from a local domain or a domain parameter d of an OObject O to an actual domain D . Each E in DE is a directed edge from a source O_{src} to a destination O_{dst} , and the label C is the class of the object being communicated. The last label is a flag, and it states whether the OEdge represents an import or an export dataflow communication. Multiple edges with different labels might exist between two OObjects.

Our analysis distinguishes between different instances of the same class C that are in different domains, even if created at the same `new` expression. In addition, the analysis treats an instance of class C with actual parameters \overline{p} differently from another instance that has actual parameters \overline{p}' . Hence, the data type of an OObject uses $C \langle \overline{D} \rangle$ instead of just a type and an owning ODomain. We follow the FDJ convention and consider an OObject's owning ODomain as

$G \in \text{OGraph}$	$::= \langle \mathbf{Objects} = DO, \mathbf{Domains} = DD, \\ \mathbf{Edges} = DE \rangle$
$D \in \text{ODomain}$	$::= \langle \mathbf{Id} = D_{id}, \mathbf{Domain} = C::d \rangle$
$O \in \text{OObject}$	$::= \langle \mathbf{Type} = C \langle \overline{D} \rangle \rangle$
$E \in \text{OEdge}$	$::= \langle \mathbf{From} = O_{src}, \mathbf{To} = O_{dst}, \\ \mathbf{Class} = C, \mathbf{Flag} = \text{Imp} \mid \text{Exp} \rangle$
DD	$::= \emptyset \mid DD \cup \{ (O, C::d) \mapsto D \}$
DO	$::= \emptyset \mid DO \cup \{ O \}$
DE	$::= \emptyset \mid DE \cup \{ E \}$
Υ	$::= \emptyset \mid \Upsilon \cup \{ C \langle \overline{D} \rangle \}$

Figure 4: Data type declarations for the OGraph.

the first element D_1 of \overline{D} . As a result of the aliasing precision provided by ownership domains, our analysis avoids merging objects excessively. It only merges two objects of the same class if all their domains are the same. The context Υ records the combination of class and domain parameters $C \langle \overline{D} \rangle$ analyzed in the call stack to avoid non-termination of the analysis due to recursive calls.

In addition to the OEdges that have OObjects as source and destination, the OGraph has ownership edges. The OGraph representation is well-formed with respect to the ownership relations declared in the code using the annotations. An ownership edge states that an OObject O is in D_1 , or that O owns a domain D . The OGraph captures this hierarchy using the DD map. Given a mapping $\{(O, C'::d) \mapsto D\}$ in DD , D is a child of O , i.e., $O \rightarrow D$. Since domains are inherited across classes [4], the class C of O can be a subclass of C' where d is declared. In the presence of recursive types, ownership edges may create cycles.

To invoke the analysis, a developer picks a root class, which is instantiated into a root object. The root class can take only one domain parameter to represent the owning domain. Typically, the root object is in the global ODomain D_{SHARED} , the root of the OGraph.

Although a domain d is declared by class C , each instance of C gets its own runtime domain $\ell.d$. For example, if there are two distinct object locations ℓ and ℓ' of class C , then the analysis distinguishes between $\ell.d$ and $\ell'.d$. Since an ODomain represents a runtime domain $\ell_i.d_i$, one domain declaration d in the code can create multiple ODomains D_i in the OGraph. We qualify a domain d by the class that declares it, as $C::d$. Since no class declares the SHARED domain, we qualify it as $::\text{SHARED}$.

Notation We use $DD[(O, p)]$ to denote a lookup of the pair (O, p) in DD . Also, by using $(O, C::d) \mapsto D \subseteq DD$, we denote that a new entry is added in DD if the map does not have an entry for the given pair. Otherwise, the above constraints is true, and DD remains unchanged.

Static Semantics. We formalize our static analysis using a constraint-based specification, as a set of inference rules, then prove that the OGraph is sound, i.e., it has all the required OObjects, ODomains, and OEdges.

In this context, soundness means that we can build a map between a ROG and an OGraph. Soundness consists of object soundness and edge soundness. With object soundness, every runtime object maps to a unique representative OObject in the OGraph. With object soundness, every runtime edge maps to a unique representative OEdge in the

OGraph. To build the maps, we instrument the FDJ dynamic semantics. Due to space limit, we present only the most interesting rules from the static semantics that create the OGraph. The instrumented dynamic semantics and the soundness proof are in the technical report [25].

In FDJ, a program is a tuple (CT, e) that consists of a class table CT , which maps classes to their definitions, and an expression e . Our analysis starts with a root expression e_{root} , that explicitly instantiates the root class C_{root} . The analysis result is the least solution $G = \langle DO, DD, DE \rangle$ of the following constraint system:

$$\emptyset, \emptyset, DO, DD, DE \vdash (CT, e_{\text{root}})$$

The analysis creates the OObject O_{root} and its owning ODomain D_{SHARED} ,

$$D_{\text{SHARED}} = \langle D_s, ::\text{SHARED} \rangle \quad O_{\text{root}} = \langle O_r, C_{\text{root}} \langle D_{\text{SHARED}} \rangle \rangle$$

then abstractly interprets e_{root} in the context of O_{root} :

$$\emptyset, \emptyset, DO, DD, DE \vdash_{O_{\text{root}}} e_{\text{root}}$$

The judgement form for expressions is as follows:

$$\Gamma, \Upsilon, DO, DD, DE \vdash_{O, H} e$$

The O subscript on the turnstile captures the context-sensitivity, and represents the context object that the analysis uses to abstractly interpret e . The H subscript is a map used by the dynamic semantics and the store typing rule in the static semantics (not shown). For readability, we omit H when not in use. $CT(C)$ and $CT(\text{Object})$ represent a lookup of a class C and the class Object in the class table, and is an implicit clause in all the static rules. (We list these clauses once at the top of Fig. 5 to avoid repetition.)

In DF-NEW, the analysis interprets a new object allocation in the context of O . The analysis first ensures that DO contains an OObject O_C for the newly allocated object. Then, DF-NEW ensures that DD has a representative ODomain D_i for each domain parameter p_i passed to the constructor of the class C . Based on the binding of each formal domain parameter α_i to actual p_i , DD maps each α_i to a corresponding D_i in the context of O_C ($(O_C, \alpha_i) \mapsto D_i$) (Fig. 5).

Then, DF-NEW uses the auxiliary judgement AUX-DOM to ensure that DD has an ODomain corresponding to each domain that C locally declares ($(O_C, C::d_j) \mapsto D_j$). AUX-DOM recursively includes inherited domains from base classes as well. AUX-OBJ1, the base case of the recursion, deals with the class Object, for which AUX-OBJ1 does nothing, because Object has no fields, domains, or methods in FDJ.

DF-NEW then obtains each expression e_R in each method m of C , and recursively processes e_R in the context of the new OObject O_C . To avoid infinite recursion, before DF-NEW analyzes e_R , it checks if the combination of the class C and actual domains \overline{D} have been previously analyzed by looking for this combination in Υ . If this combination does not exist, DF-NEW extends Υ with the current combination. As a side note, Υ tracks previously analyzed OObjects only at the call stack level. It does not do so globally across the program because similar combinations of the same class and domain parameters can occur in different contexts, and must be analyzed separately. Finally, DF-NEW analyzes each argument of the constructor. Since our analysis distinguishes between a field initialization in a constructor and a field write, DF-NEW does not require dataflow edges in DE .

$$CT(C) = \text{class } C \langle \bar{\alpha}, \bar{\beta} \rangle \text{ extends } C' \langle \bar{\alpha} \rangle \{ \bar{T} \bar{f}; \bar{dom}; \dots; \bar{md}; \} \quad CT(\text{Object}) = \text{class Object} \langle \alpha_o \rangle \{ \}$$

$$\begin{array}{c}
\forall i \in 1..|\bar{p}| \quad D_i = DD[(O, p_i)] \quad \text{params}(C) = \bar{\alpha} \\
O_C = \langle C \langle \bar{D} \rangle \rangle \quad \{O_C\} \subseteq DO \quad \alpha_i \in \bar{\alpha} \\
\{(O_C, \alpha_i) \mapsto D_i\} \subseteq DD \quad \{(O_C, p_i) \mapsto D_i\} \subseteq DD \\
DO, DD, DE \vdash_O \text{ddomains}(C, O_C) \\
\forall m \in \bar{md} \text{ mbody}(m, C \langle \bar{p} \rangle) = (\bar{x} : \bar{T}, e_R) \\
C \langle \bar{D} \rangle \notin \Upsilon \implies \{\bar{x} : \bar{T}, \text{this} : C \langle \bar{p} \rangle\}, \Upsilon \cup \{C \langle \bar{D} \rangle\}, DO, DD, DE \vdash_{O_C} e_R \\
\Gamma, \Upsilon, DO, DD, DE \vdash_O \bar{e} \\
\hline
\Gamma, \Upsilon, DO, DD, DE \vdash_O \text{new } C \langle \bar{p} \rangle (\bar{e}) \quad \text{[DF-NEW]}
\end{array}$$

$$\begin{array}{c}
\forall (\text{domain } d_j) \in \bar{dom} \quad D_j = \langle D_{id_j}, C::d_j \rangle \quad \{(O_C, C::d_j) \mapsto D_j\} \subseteq DD \\
DO, DD, DE \vdash_O \text{ddomains}(C', O_C) \\
\hline
DO, DD, DE \vdash_O \text{ddomains}(C, O_C) \quad \text{[AUX-DOM]}
\end{array}$$

$$\begin{array}{c}
O_k \in DO \quad O_k = \langle C \langle \bar{D} \rangle \rangle \quad C <: C' \\
\forall i \in 1..|\bar{p}'| \quad D'_i = DD[(O, p'_i)] \quad D'_i = D_i \\
\hline
DO, DD, DE \vdash_O \text{lookup}(C' \langle \bar{p}' \rangle) = \{O_k\}_{k \in 1..sz} \quad \text{[DF-LOOKUP]}
\end{array}$$

$$\begin{array}{c}
e_0 : C \langle \bar{p} \rangle \quad (T_k \ f_k) \in \text{fields}(C \langle \bar{p} \rangle) \\
DO, DD, DE \vdash_O \text{import}(C \langle \bar{p} \rangle, T_k) \\
\Gamma, \Upsilon, DO, DD, DE \vdash_O e_0 \\
\hline
\Gamma, \Upsilon, DO, DD, DE \vdash_O e_0.f_k \quad \text{[DF-READ]}
\end{array}$$

$$\begin{array}{c}
e_0 : C \langle \bar{p} \rangle \quad (T_k \ f_k) \in \text{fields}(C \langle \bar{p} \rangle) \\
e_1 : C_1 \langle \bar{p}'' \rangle \quad C_1 \langle \bar{p}'' \rangle <: T_k \\
DO, DD, DE \vdash_O \text{export}(C \langle \bar{p} \rangle, C_1 \langle \bar{p}'' \rangle) \\
\Gamma, \Upsilon, DO, DD, DE \vdash_O e_0 \quad \Gamma, \Upsilon, DO, DD, DE \vdash_O e_1 \\
\hline
\Gamma, \Upsilon, DO, DD, DE \vdash_O e_0.f_k = e_1 \quad \text{[DF-WRITE]}
\end{array}$$

$$\begin{array}{c}
DO, DD, DE \vdash_O \text{lookup}(T_{src}) = \{O_i\}_{i \in 1..sz} \\
DO, DD, DE \vdash_{O_i} \text{lookup}(T_{label}) = \{O_j\}_{j \in 1..sz'} \\
\forall i \in 1..sz \ \forall j \in 1..sz' \ O_j = \langle C_j \langle \bar{D} \rangle \rangle \quad \{\langle O_i, O, C_j, \text{Imp} \rangle\} \subseteq DE \\
\hline
DO, DD, DE \vdash_O \text{import}(T_{src}, T_{label}) \quad \text{[AUX-IMPORT]}
\end{array}$$

$$\begin{array}{c}
DO, DD, DE \vdash_O \text{lookup}(T_{dst}) = \{O_i\}_{i \in 1..sz} \\
DO, DD, DE \vdash_O \text{lookup}(T_{label}) = \{O_j\}_{j \in 1..sz'} \\
\forall i \in 1..sz \ \forall j \in 1..sz' \ O_j = \langle C_j \langle \bar{D} \rangle \rangle \quad \{\langle O, O_i, C_j, \text{Exp} \rangle\} \subseteq DE \\
\hline
DO, DD, DE \vdash_O \text{export}(T_{dst}, T_{label}) \quad \text{[AUX-EXPORT]}
\end{array}$$

$$\begin{array}{c}
e_0 : C \langle \bar{p} \rangle \quad \text{mtype}(m, C \langle \bar{p} \rangle) = \bar{T} \rightarrow T_R \\
DO, DD, DE \vdash_O \text{import}(C \langle \bar{p} \rangle, T_R) \\
\forall k \in 1..|\bar{e}| \ e_k : T'_k \quad T'_k <: T_k \quad T_k \in \bar{T} \quad DO, DD, DE \vdash_O \text{export}(C \langle \bar{p} \rangle, T'_k) \\
\Gamma, \Upsilon, DO, DD, DE \vdash_O e_0 \quad \Gamma, \Upsilon, DO, DD, DE \vdash_O \bar{e} \\
\hline
\Gamma, \Upsilon, DO, DD, DE \vdash_O e_0.m(\bar{e}) \quad \text{[DF-INVK]}
\end{array}$$

Figure 5: Static semantics. Additional rules (Df-Var, Df-Loc, Df-Context, Df-Sigma) are in [25].

DF-LOOKUP defines the auxiliary judgement *lookup* that returns the set of the OObjects O_k in DO such that the class of O_k is C' or one of its subclasses. It also ensures that each domain D_i of O_k corresponds to D'_i , a domain associated with O in DD . The second condition increases the precision of our analysis, because *lookup* returns only a subset of all the objects of class C' or its subclasses in DO . From this subset, our analysis picks the source or destination OObjects, and finds the class representing the label of an OEdge.

The auxiliary judgements AUX-IMPORT and AUX-EXPORT ensure import and export edges between the context OObject O and the OObjects O_i , where O_i is the result of *lookup* (T_{src}), and *lookup* (T_{dst}), respectively. The direction of the edge is from O_i to the context O for AUX-IMPORT, and from the context O to O_i for AUX-EXPORT. To identify an edge's label, AUX-EXPORT calls *lookup* in the context of O , while AUX-IMPORT calls the second *lookup* in the con-

text of O_i . As a result, there could be multiple edges with different labels between the same two OObjects, depending on what *lookup* returns.

DF-READ and DF-WRITE abstractly interpret field read and field write expressions, respectively, and use AUX-IMPORT and AUX-EXPORT. Both auxiliary judgements take the type e_0 as the first argument, and pass it to *lookup* to set the source and destination OObjects. For the label, DF-READ uses the type of the field f_k , while DF-WRITE uses the type of the right-hand side expression e_1 . The labels are the classes of these types or one of their subclasses.

DF-INVK abstractly interprets method invocation expressions. First, it ensures the existence of an import edge from the receiver of the method to the context OObject O . The label of the import edge is the class of the return type, or one of its subclasses. Next, for each argument e_k , DF-INVK ensures the existence of an export edge from O to the re-

ceiver of the method. The label of each export edge is the class of the argument or one of its subclasses. The rule ensures export edges only for a method invocation with at least one argument. Finally, the rule evaluates recursively the expressions e_0 and \bar{e} .

Recursive Types. The analysis must handle recursive types which lead to an unbounded number of nodes in the OGraph. To get a finite OGraph and ensure that the analysis terminates, the analysis could stop expanding an OGraph after a certain depth. However, truncating the recursion at an arbitrary depth may omit objects or edges beyond the cutoff depth. Instead, the analysis creates a cycle in the OGraph when it encounters a domain declaration $C::d$ already analyzed in the context of the OObject O_C . According to DF-NEW, a domain declaration $C::d$ may be analyzed multiple times using the context O_C . If the context is different, the analysis creates multiple ODomains for the same domain declaration d . In the presence of a recursive type, the pair $(O_C, C::d)$ will be the same for multiple passes of the analysis. On the first pass, the analysis adds $C<\bar{D}>$ of O_C in Υ and performs a lookup in DD . If the pair is not found, it creates a new ODomain and a new entry for the pair in DD . On the second pass, the analysis encounters the same pair and reuses the existing ODomain. So the analysis creates a cycle in the OGraph, and the reused ODomain appears as the child of two OObjects. This justifies an ODomain not having a unique owning OObject (Fig. 4). On the next pass, the analysis encounters the same $C<\bar{D}>$ in Υ and does not recurse any further. For further discussion of recursion, refer to the technical report [25].

Differences with points-to analysis. Our formalization is similar to the one for the points-to analysis [1, Section 3.2 and 3.3]. The two analyses create the same object-domain hierarchy, but the analysis in this paper shows additional edges that are missing from an OOG with points-to edges. The key differences in the formalization deal with generating the dataflow edges and the soundness proof. The previous work made a simplistic assumption about dataflow edges, namely that they can be approximated by reverting points-to edges, but the assumption turned out to be imprecise.

5. EVALUATION

We implemented our analysis and applied it to an extended example to evaluate the research hypothesis.

Hypothesis: *Given legacy code to which we add ownership domain annotations, a static analysis can extract an OOG depicting dataflow edges that correspond to those manually drawn by a developer who is reasoning about the runtime architecture of a system and dataflow communication.*

Research Questions. To evaluate the hypothesis, we formulated the following research questions:

RQ1: Are the extracted dataflow edges consistent with the edges on a documented runtime architecture drawn by a security expert?

RQ2: Are the extracted dataflow edges different from the points-to edges in an OOG?

Comparable diagrams. For the evaluation, we looked for documented runtime architectures that depict dataflow communication. Ideal candidates are DFDs that are used in architectural risk analysis. DFDs are also hierarchical. For example, a Level-1 DFD (L1) shows the main components of the system. A Level-2 (L2) DFD shows the components of

Table 1: Mapping between DFD and OOG.

Component	OObject
Crypto Consumer	mgr:CustomerManager
Protected Data	cust:CustomerInfo; cci:CreditCardInfo; plaintexts:DecryptionResults
Crypto Provider	provider:Provider
Engine	engine:EngineWrapper
Key Manifest	alias:KeyAlias
Key Vault	keyStore:LocalKeyStore
Key Manager	keyTool:KeyTool
Receipts	cryptoReceipt:CryptoReceipt
Encoder	encoder:Utils

an L1 DFD broken into their constituent sub-components. **Subject System.** For the subject system, we selected CryptoDB, a secure database system designed by a security expert [10]. This system is implemented in Java and consists of 3,000 lines of code. The documentation includes a DFD-L1 (Fig. 6(a)) and a DFD-L2 (Fig. 6(c)) drawn by the security expert. The presence of both a Java implementation and DFDs make CryptoDB an appropriate choice to evaluate our approach. We used these DFDs as a guidance while adding the annotations in the code [1, Section 7.8].

Tools. We extracted two OOGs, one with dataflow edges and one with points-to edges. The tool allowed us to control the level of detail in the OOG by collapsing the substructure of selected objects and by automatically lifting. Also, since our analysis can extract multiple dataflow edges with different labels between the same source and destination, we post-processed the OOG to show only one edge having the merged set of labels (Fig. 6).

Extracted vs. manually drawn edges. We compared DFD-L1 against the OOG where all the objects have their substructure collapsed (OOG-L1) (Fig. 6(b)). As the success criteria for RQ1, we considered both the direction of the edges and the edge labels. DFD-L1 included 9 components and showed how Protected Data passed from a Crypto Consumer to a Crypto Provider and then to an encryption Engine. The OOG-L1 also had 9 objects, but the component-object mapping was not straightforward.

Mapping between DFD and OOG. The mapping between the components in a DFD and the objects in the OOG (Table 1) helps explain the correspondence between the edges. For example, we mapped Crypto Consumer and Crypto Provider to mgr and provider, respectively. DFD-L2 shows the sub-components of Crypto Consumer and Crypto Provider; in the OOG, we expanded the substructure of mgr and provider (OOG-L2) (Fig. 6(d)).

RQ1 Procedure. To compare a DFD to an OOG with dataflow edges, we did the following steps: (1) we extracted the OOG and collapsed it until it shows the same level of abstraction as the DFD; (2) using the mapping between names, we compared the components in the DFD and the objects in the OOG; (3) we compared the DFD edges to the dataflow edges by visual inspection; (4) if an edge appeared in both the DFD and the OOG, we checked if the OOG edge labels can be recognized in the DFD edge labels; (5) if an edge appeared in the OOG but was missing from the DFD, we traced from the edge in the OOG to the corresponding lines of code to confirm that the edge was not a false positive.

The OOG and the DFD had a relatively low number of nodes so we used visual inspection. For larger OOGs, we

Table 2: Labels for the numbered edges in Fig. 6.

	OEdge labels (merged)	DFD Label
1	CustomerInfo; CreditCardInfo; CompoundCryptoReceipt; CryptoReceipt	Unencrypted Data
2	DecryptionResults; CompoundCryptoReceipt;	Encrypted Data
3	LocalKey	Key Data
4	LocalKey; String	Task Feedback

Table 3: Points-to vs. dataflow edges.

	PtE	DE	$DE \setminus PtE$	$PtE \setminus DE$	$DE \cap PtE$
OOG-L1	6	19	13	1	5
OOG-L2	7	24	20	3	4

could use the automated tools to compare the extracted diagram to the documented one [2].

Observation 1: A cycle of dataflow edges drawn by developers existed also in OOG-L1. The DFD had a cycle between 5 components (Fig 6(a)). A cycle was also present between the corresponding objects in the OOG: `provider` \rightarrow `engine` \rightarrow `keyStore` \rightarrow `keyTool` \rightarrow `alias` \rightarrow `provider`. The same DFD components were also linked by a second cycle in the opposite direction. In the OOG, the edge `provider` \rightarrow `alias` was missing. The inspection of the `Provider` class revealed that the edge was missing due to the limitations of our analysis that did not handle static code.

Observation 2: The OOG edge labels were recognized in the corresponding DFD edge labels. We show the most interesting edge labels in Fig. 6(b) listed in Table 2. DFD-L1 showed an edge from `Key Vault` to `Engine` labeled `Key Data`, while the corresponding edge in OOG-L1 from `engine` to `keyStore` had the label `LocalKey` (3) i.e., `LocalKey` was recognized as `Key Data`. While “Data” is a general term, the `LocalKey` indicated “what kind” of data `keyStore` provided. As another example, DFD-L1 showed the edge label `Unencrypted Data` from `Crypto Consumer` to `Crypto Provider`. The corresponding labels in the OOG were `CustomerInfo`, `CreditCardInfo`, and `CompoundCryptoReceipt` (1). The opposite edge in DFD-L1 showed that the `Encrypted Data` corresponded `CompoundCryptoReceipt` and `DecryptionResults` (2). The inspection of `CryptoReceipt` and `CompoundCryptoReceipt` revealed that these classes implemented encrypted data, while `CustomerInfo` and `CreditCardInfo` represented unencrypted data. The edge label `DecryptionResults` from `provider` to `mgr` indicated that `provider` was also responsible for decryption. This information was not obvious in DFD-L1, while in DFD-L2 the corresponding edge labels `Data` and `Result` are very general.

RQ2 Procedure. To compare an OOG with points-to edges against an OOG with dataflow edges, we followed the next steps: (1) we found the points-to edges without a corresponding dataflow edge; we traced dataflow edges to the appropriate lines of code to confirm the absence of communication; (2) we found the dataflow edges without a corresponding points-to edge; we traced dataflow edges to the appropriate lines of code to confirm the absence of points-to edges; (3) we repeated the procedure for both OOGs.

Observation 3: Dataflow edges are different from points-to edges. We compared the set of dataflow edges (DE) against the set of points-to edges (PtE) in OOG-L1 and OOG-L2 (Table 3). The two sets are different, with little overlap between dataflow and points-to edge. In the OOG

with points-to edges only, the `alias` object was isolated, and the cycle formed by dataflow edges was missing. Such an OOG might be unsuitable to reason about communication between objects. We inspected the cases when objects were connected through dataflow edges only. In these cases, a method might create an object, without storing it in a field as a persistent relation. We found for example that `CustomerManager` had a method `getCustomer(String)` which instantiated the `cci` and `cust` objects, read the expiration date from the database and passed `cci` to the `cust` object, which was returned. `CustomerManager` had only one field of type `Provider`, so the analysis did not create points-to edges from `mgr` to `cci` or `cust`.

Observation 4: By expanding an OOG, we can recognize the sub-components in DFD-L2. We expanded the substructure of `mgr` and `provider`, and the OOG showed the `OObjects` `cryptoReceipt` in the substructure of `mgr`, and `receipts` and `encoder` in the substructure of `provider` (Fig. 6(d)). In DFD-L2, the compound processes `Crypto Consumer` and `Crypto Provider` were broken into the sub-components: `Receipts`, `Business Logic`, `Receipt Manager`, `Encoder`, and `Initializer`. The OOG showed the bidirectional communication between `provider` and `receipts`, also shown in DFD-L2 as a `Request - Result` communication between `Crypto Provider` and `Receipt Manager`. We also found several divergences and absences in the OOG. For example, DFD-L2 shows `Business Logic` as a sub-component of `Crypto Consumer`; however, they were not clearly separated in the implementation. Similarly, `Initializer` was not clearly separated from `Crypto Provider`. Therefore, these components were absent in the OOG. As an example of a divergence, OOG-L2 showed a dataflow edge from `cryptoReceipt` to `receipts`, while in DFD-L2 there was no edge between the corresponding components `Receipts` and `Receipt Manager`. Overall, by expanding the OOG, we observed that the implementation did not follow strictly the design. Indeed, the book confirms that the implementation was intended to show the primary functionality of the components, while secondary functionalities were glossed over [10, Chapter 13].

Observation 5: Rarely do import dataflow edges overlap with export edges. OOG-L1 showed only one overlap between an import and export edges, while OOG-L2 showed no overlap. In OOG-L1, the overlap edge `mgr` \rightarrow `provider` was a lifted edge. Instead of this edge, the OOG-L2 showed two import edges: `cryptoReceipt` \rightarrow `provider` and `cryptoReceipt` \rightarrow `receipts`. We traced these edges to the code and we found that both `provider` and `receipts` read the encoded ciphertext represented as a `String`.

Threats to validity. Our evaluation was on only one system. We can mitigate this threat to validity by extracting OOGs with dataflow edges for additional systems. We already have several annotated systems totalling 68 KLOC [1], from which we extracted OOGs with points-to edges. We can reuse these annotated systems to extract dataflow edges. However, some of these systems do not have a documented runtime architecture that we can compare against.

There are a few expressiveness challenges of the ownership domain type system. For example, ownership type systems struggle with static code, which resulted in several absent edges in OOG-L2. Also, ownership domains have special annotations that require an additional analysis to resolve them. For example, the `lent` annotation indicates a temporary reference to an object. If the receiver of a method

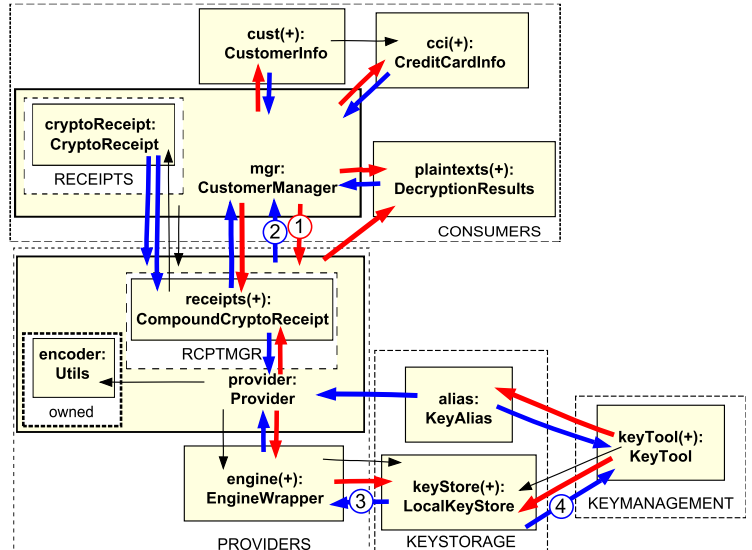
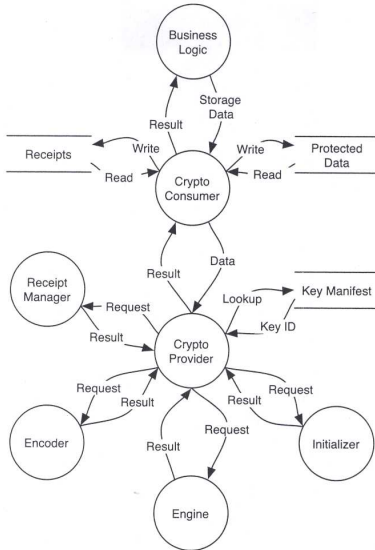
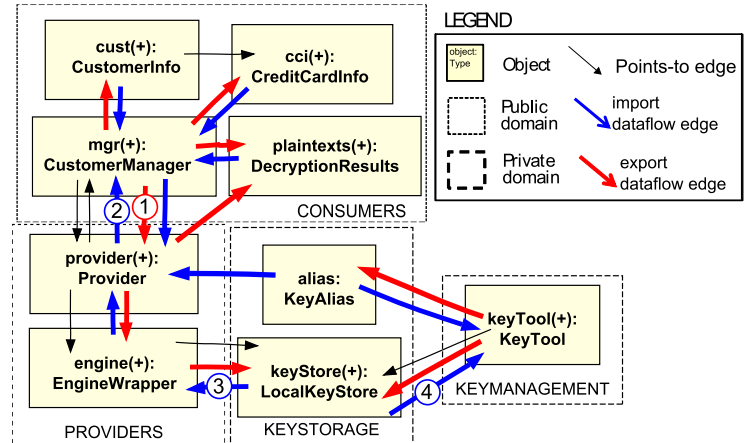
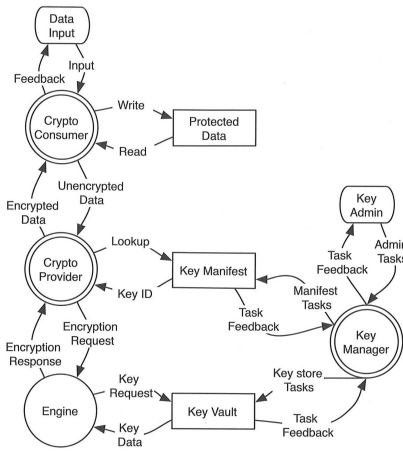


Figure 6: Documented DFDs vs. automatically extracted OOG in CryptoDB.

is annotated with `lent`, the additional analysis needs to determine the actual domain the receiver is in. Currently, the analysis omits the corresponding dataflow edges. In CryptoDB, only 7% of the annotations were `lent`, so they had a limited influence on the results. Currently, one workaround is to manually resolve the `lent` annotation, and to use a more precise annotation.

6. RELATED WORK

Our work focuses on extracting one type of information, namely usage relationships, i.e., an object is using a method or a field of another object. We discuss how the related analyses address the challenges we listed above (Section 3). **Dataflow Communication.** Andersen’s static analysis extracted dataflow and points-to information from programs written in C [5], and was extended to object oriented code [15, 23] including Java [20]. These analyses determined the memory locations that may be modified by the execution of a statement. A dataflow edge means that *an object a owns a reference to an object c, and passes it to an object b, or an object a owns a reference to an object b, from*

which it receives a reference to an object c which only b knew before [20]. However, the results of these analyses are flat graphs [20, 9] and the analyses did not attempt to be sound. In contrast, our analysis extracts hierarchical object graphs, with a relatively small number of top-level objects [1].

Sensitivity. An object graph analysis can be flow-, context-, or object-sensitive. A flow-sensitive analysis considers the order in which methods are called. A context-sensitive analysis analyzes the methods for each context under which a method is invoked. Object-sensitive analyses for points-to and dataflow edges addressed the aliasing and precision challenges [23, 15]. However, the analysis might not scale for a large number of references. Such an analysis worked well for on-demand based approaches which refined the references analyzed [21]. Seeking a tradeoff between soundness and precision, our analysis considers ownership domains as contexts and distinguishes objects of the same type but in different domains. That is, our analysis is *domain-sensitive*, and object- and flow-insensitive.

Dynamic analyses. Object graphs were extracted by analyzing heap snapshots [16, 17], and execution traces [13].

Lienhard analyzed execution traces and extracted an Object Flow Graph (OFG) in which edges represent objects, and nodes represent code structures: classes, and groups of classes [13]. OFG analysis addressed aliasing challenge, and linked objects to field read, field write, and method invocation expressions in the code, the same expressions used by our analysis. Since one class corresponds to one OFG node, an OFG is unable to show the communication between different instances of the same class and does not meet the soundness challenge. One advantage of dynamic analysis is that it does not require annotations. However, it can only infer a strict, owner-as-dominator hierarchy, which is limited in representing some design idioms [4]. Ownership domains support both strict encapsulation and logical containment (through public domains) and thus can express arbitrary design intent without restricting accessibility. In addition, a dynamic analysis requires extensive graph summarization to obtain an abstracted object graph [17, 8].

Annotation-based static analyses. Lam and Rinard [12] proposed a type system and a static analysis where by developer-specified annotations guide the static abstraction of an object model by merging objects based on tokens. Their approach supports a fixed set of statically declared global tokens, and their analysis shows a graph indicating which objects appear in which tokens. Since there is a statically fixed number of tokens, all of which are at the top level, an extracted object model is a top-level architecture that does not support hierarchical decomposition, thus limiting the scalability of the object model. In addition to their object model, Lam and Rinard extract models for “sub-system access”, “call/return interaction”, and “heap interaction”, which is similar to the dataflow information our analysis extracts. From the challenges we listed in Section 3, they addressed aliasing, summarization in the presence of recursive types, and precision supported by tokens. Our approach extends Lam and Rinard’s both to handle hierarchical object graphs and to support object-oriented language constructs such as inheritance.

7. CONCLUSION

We proposed a static analysis to extract a hierarchical object graph with dataflow communication edges that show usage relations between objects. We formalized the analysis following ownership domains and Featherweight Domain Java, and proved its soundness. We evaluated our analysis on an extended example and showed that the dataflow edges extracted by our analysis are similar to the ones drawn by developers who are reasoning about security.

8. REFERENCES

- [1] M. Abi-Antoun. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure*. PhD thesis, CMU, 2010.
- [2] M. Abi-Antoun and J. Aldrich. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *OOPSLA*, 2009.
- [3] M. Abi-Antoun and J. M. Barnes. Analyzing Security Architectures. In *ASE*, 2010.
- [4] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.
- [5] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
- [6] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architecture: View and Beyond*. Addison-Wesley, 2003.
- [7] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *TSE*, 35(4), 2009.
- [8] T. Hill, J. Noble, and J. Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Visual Lang. and Comp.*, 13(3), 2002.
- [9] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 27(2), 2001.
- [10] K. Kenan. *Cryptography in the Database*. Addison-Wesley, 2006. Code: http://kevinkenablogs.com/downloads/cryptodb_code.zip.
- [11] R. Koschke. Architecture Reconstruction: Tutorial on Reverse Engineering to the Architectural Level. In A. D. Lucia and F. Ferrucci, editors, *International Summer School on Software Engineering*, 2008.
- [12] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, 2003.
- [13] A. Lienhard, S. Ducasse, and T. Gîrba. Taking an object-centric view on dynamic information with object flow analysis. *Journal of Computer Languages, Systems and Structures (COMLAN)*, 35:63–79, 2009.
- [14] G. McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.
- [15] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-To Analysis for Java. *TOSEM*, 14(1), 2005.
- [16] N. Mitchell. The Runtime Structure of Object Ownership. In *ECOOP*, 2006.
- [17] N. Mitchell, E. Schonberg, and G. Sevitsky. Making Sense of Large Heaps. In *ECOOP*, 2009.
- [18] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *TSE*, 27(4), 2001.
- [19] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *TSE*, 32(7), 2006.
- [20] A. Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FU Berlin, 2002.
- [21] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *OOPSLA*, 2005.
- [22] F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft Press, 2004.
- [23] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code*. Springer-Verlag, 2004.
- [24] P. Torr. Demystifying the Threat-Modeling Process. *IEEE Security and Privacy*, 3(5), 2005.
- [25] R. Vanciu and M. Abi-Antoun. Extracting Dataflow Communication from Object-Oriented Code. Technical report, WSU, 2011. www.cs.wayne.edu/~mabianto/tech_reports/VA11_TR.pdf.